Semantic Analyses for Storage Management Optimizations in Functional Language Implementations

by

Young Gil Park

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
September 1991

Approved : _____

Benjamin Goldberg
Research Advisor

©Young Gil Park
All Rights Reserved. 1991

To
Grace Sunjung and Jihwa Park

Acknowledgements

First, I would like to express my sincere thanks to my advisor, Professor Benjamin Goldberg, for all his guidance, suggestions, advice and the many other things throughout the years. I would also like to thank Professors Robert Paige and Edmond Schonberg for reading a draft of this thesis and giving valuable comments as well as serving on my thesis committee. I wish to thank all the other members of my committee, Professors Robert Dewar and Malcolm Harrison, for their helpful comments and encouragements.

For their advice and help, I am grateful to all the people in Programming Languages Lab at Courant Institute, Professors Paul Hudak at Yale University, Arvind at MIT, Peter Lee at Carnegie-Mellon University and Dr. Henry Baker at Nimble Computer Corporation.

Last, by no means least, I would like to thank my wife Jihwa and my daughter Grace Sunjung - I owe everything to you! I would also like to thank my parent for their constant love and support.

This research was supported in part by Korea Government Graduate Fellowship, National Science Foundation (#CCR-8909634) and DARPA/ONR (#N00014-90-1110).

Abstract

Semantic Analyses for Storage Management Optimizations
in Functional Language Implementations

Young Gil Park

Ph.D., Department of Computer Science, New York University, September 1991.

(Research advisor: Benjamin Goldberg)

One of the major overheads in implementing functional languages is the storage management overhead due to dynamic allocation and automatic reclamation of indefinite-extent storage. This dissertation investigates the problems of statically inferring lifetime information about dynamically-allocated objects in *higher-order polymorphic* functional languages, both *strict* and *non-strict*, and of applying that information to reduce the storage management overhead.

We have developed a set of compile-time semantic analyses for a higher-order, monomorphic, strict functional language based on denotational semantics and abstract interpretation. They are 1) *escape analysis*, which provides information about the relative lifetimes of objects such as arguments and local objects defined within a function with respect to an activation of the function call, 2) *refined escape analysis* which, as a refinement of escape analysis, provides information about the lifetimes of components of aggregate structures, and 3) *reference escape analysis* which provides information about the relative lifetimes of references created within a function with respect to an activation of the function.

We also have developed a compile-time semantic analysis called *order-of-demand analysis* for higher-order, monomorphic, non-strict functional languages, which provides information about the order in which the values of bound variables are demanded and thus allows one to compute a range of information including strictness, evaluation-order, and evaluation-status information.

Using the notion of *polymorphic invariance*, we describe a method for analyzing a polymorphic language by using the analyses for a monomorphic language. We then extend those analyses for a strict language to a non-strict language using non-strict program transformation and evaluation-status information.

Based on statically inferred escape information, we propose a combination of storage management optimization techniques including stack allocation, explicit reclamation, in-place reuse, reference counting elimination, block allocation/reclamation, and improving generational garbage collection.

Contents

1 Introduction **1**

1.1 Functional Languages 1
1.2 Storage Management Overhead 4
1.3 The Role of Lifetime Information 6
1.4 Lazy Evaluation Overhead and the Role of Order-of-Demand Information . 8
1.5 Semantics-based Analysis 9
1.6 Overview of the Thesis 11
1.6.1 The Functional Language 12
1.6.2 Strict and Non-strict Standard Semantics 13
1.6.3 Organizations 14

2 Escape Analysis **18**

2.1 Escapement of Objects 19
2.2 Exact Non-standard Escape Semantics 21
2.3 Abstract Escape Semantics 25
2.4 Escapement Testing 33
2.5 Improving Precision of Escapement 38
2.5.1 Positions of a List 38
2.5.2 Improved Abstract Escape Semantics 39
2.5.3 Improved Escapement Testing 44
2.6 Complexity of Escape Analysis 47

3 Refinements of Escape Analysis **48**

3.1 Refined Escapement of Objects 49
3.2 Refined Escape Analysis 51
3.2.1 Spines of a List 51

3.2.2 Exact Refined Escape Semantics 52
3.2.3 Abstract Refined Escape Semantics 54
3.2.4 Refined Escapement Testing 59
3.3 Improving Precision of Refined Escapement 64
3.4 Comparison to Escape Analysis and Complexity 71

4 Reference Escape Analysis **73**
4.1 Escapement of References 74
4.2 Function Transformation 76
4.3 Exact Reference Escape Semantics 77
4.4 Abstract Reference Escape Semantics 80
4.5 Reference Escapement Testing 84
4.6 Improving Precision of Reference Escapement 89
4.7 Complexity of Reference Escape Analysis 96

5 Order-of-Demand Analysis **98**
5.1 Order of Demand under Lazy Evaluation 99
5.2 Before Analysis 101
5.2.1 Exact Before Semantics 102
5.2.2 Abstract Before Semantics 104
5.2.3 Testing for Before Demand 110
5.3 Using Before Analysis 113
5.3.1 Computing Status-of-Evaluation Information 114
5.3.2 Computing Order-of-Evaluation Information 114
5.3.3 Computing Strictness Information 115
5.3.4 Examples 116
5.4 Complexity of Before Analysis 119
5.5 Extensions to Other Evaluation Models 120

6 Polymorphic Invariance **122**
6.1 Issues in Analyzing Polymorphic Functions 122
6.2 Polymorphic Invariance 123
6.3 Polymorphic Invariance Proofs 124
6.3.1 Escape Analysis 124
6.3.2 Refined Escape Analysis 127
6.3.3 Reference Escape Analysis 131

6.3.4 Order-of-Demand Analysis 135
6.4 Analysis of Polymorphic Languages 138

7 Extensions to Non-strict Languages 141
7.1 Escapement under Normal-Order Evaluation 142
7.1.1 Program Transformation 144
7.1.2 Using Escape Analysis for Strict Languages 144
7.1.3 Using Reference Escape Analysis for Strict Languages 146
7.1.4 Examples 148
7.2 Escapement under Lazy Evaluation 151
7.2.1 Using Status-of-Evaluation Information 154
7.2.2 Escape Analysis for Lazy Evaluation 154
7.2.3 Reference Escape Analysis for Lazy Evaluation 157
7.2.4 Examples 160
7.3 Escapement under Evaluation with Strictness 163

8 Storage Management Optimizations 165
8.1 Stack Allocation 165
8.2 Explicit Reclamation 169
8.3 In-place Reuse 172
8.4 Reference Counting Elimination 174
8.5 Block Allocation/Reclamation 177
8.6 Improving Generational Garbage Collection 178

9 Related Work, Conclusions, and Future Work 180
9.1 Related Work 180
9.2 Summary 183
9.3 Future Work 185

List of Figures

1.1 The Syntax of Functional Language 13
1.2 Standard Semantic Functions 15

2.1 Escapement of Objects 19
2.2 The Basic Escape Domain 22
2.3 Escape Semantic Functions 26
2.4 The Abstract Basic Escape Domain 27
2.5 Abstract Escape Semantic Functions 29
2.6 Positions of a List 39
2.7 The Improved Abstract Basic Escape Domain 40
2.8 Improved Abstract Escape Semantic Functions 41
2.9 Improved Abstract Escape Semantic Functions 42
2.10 Relationship among Standard and Escape semantics 45

3.1 Refined Escapement of Objects 50
3.2 Spines of a List 51
3.3 The Basic Refined Escape Domain 53
3.4 Refined Escape Semantic Functions 55
3.5 The Abstract Basic Refined Escape Domain 56
3.6 Abstract Refined Escape Semantic Functions 58
3.7 The Improved Abstract Basic Refined Escape Domain 65
3.8 Improved Abstract Refined Escape Semantic Functions 67
3.9 Relationship among Standard and Refined Escape Semantics 69
3.10 Relationship among Escape and Refined Escape Semantics 72

4.1 Escapement of References 74
4.2 The Basic Reference Escape Domain 77
4.3 Reference Escape Semantic Functions 79

4.4 The Abstract Basic Reference Escape Domain 80
4.5 Abstract Reference Escape Semantic Functions 82
4.6 The Improved Abstract Basic Reference Escape Domain 90
4.7 Improved Abstract Reference Escape Semantic Function 92
4.8 Relationship among Reference Escape Semantics 94

5.1 The Basic Before Domain 103
5.2 Before Semantic Functions 105
5.3 The Abstract Basic Before Domain 106
5.4 Abstract Before Semantic Functions 108
5.5 Relationship among Standard and Before semantics 110

7.1 Non-strict Program Transformation 145
7.2 Abstract Escape Semantic Functions for Lazy Evaluation 155
7.3 Abstract Reference Escape Semantic Functions for Lazy Evaluation 158
7.4 Non-strict Program Transformation with Strictness Information 164

Chapter 1

Introduction

Functional programming languages provide a variety of useful features such as referential transparency, higher-order functions, non-strict semantics and implicit storage management. Functional languages, however, have gained popularity much slower than imperative ones because their implementations on conventional sequential and parallel computers have tended to show relatively poor performance. Optimization to improve the performance of functional languages is thus an essential component in any viable implementation. One of the major overheads incurred by functional language implementations is the storage management overhead due to dynamic allocation and automatic reclamation of indefinite-extent storage. This thesis explores the optimization problem of reducing the storage management overhead by obtaining lifetime information about dynamically-allocated objects in higher-order polymorphic functional languages, which is inferred at compile-time through semantics-based analyses of high-level source programs.

1.1 Functional Languages

The class of modern functional programming languages exhibits several useful features as follows ([7], [31], [40], [66]):

- *Referential Transparency (Equational Reasoning) :* In pure functional programming, a program has no modifiable state, because of the lack of an assignment operator, and thus is made up entirely of expressions. The evaluation of an expression is guaranteed to have no other effects on the program, that is, no side-effects. The absence of side-effects guarantees the mathematical property of *referential transparency* which means that any syntactically identical pair of expressions are semantically the same, scope

rules allowing, and this makes functional programs easier to reason about. Programs can be treated as a system of equations and *equational reasoning* can be applied. Functional languages also have a *completeness property* which states that the reduction of an expression using a general normal-order strategy is guaranteed to yield a result if a result is mathematically deducible.

- *Higher-Order Functions :* The notion of a function is the primary abstraction mechanism in any programming language and thus facilitating the use of functions increases the utility of that kind of abstraction. Much of the power of a programming language can come from the advanced use of functions. Imperative languages such as Algol68, Pascal and C allow procedures or functions to be passed as parameters to other procedures and functions, but does not permit function-valued or procedure-valued functions. In functional languages, functions are treated as *first-class* objects like any other data object in the language. Functions that are allowed to be put in data structures, passed as an argument in function calls, and returned as values from expressions including function calls are called *first-class* functions. Those languages supporting higher-order functions generally allow functions to be *curried* or *partially applicable*. If a function is defined to take several arguments, it can be considered as a function that takes only one argument and returns a curried function.

- *Non-strict Semantics :* The reduction of a function application in a language can generally be performed in two ways: strict (applicative-order) evaluation order and non-strict (normal-order) evaluation. In *strict semantics*, the application of a function to its arguments results in the arguments being evaluated before they are passed to the function and before the body of the function begins execution. This corresponds to *applicative-order reduction* in the lambda calculus. The advantage of this scheme, known as *call-by-value*, is that it is easy to implement efficiently; first we evaluate the arguments, then we call the function. However, it may result in unnecessary evaluation if an argument's value is not ultimately required by the called function. In *non-strict semantics*, an argument is not evaluated unless and until its value is demanded inside the body of the called function; all arguments are passed to the function in an unevaluated form and are only evaluated when needed inside the function body. This corresponds to *normal-order reduction* in the lambda calculus. The advantage of this evaluation method, also known as *call-by-name*, is that no effort will have been wasted if the argument value is not ultimately required. A non-strict semantics also has the advantage of resulting in program termination more often than a strict

semantics since the evaluation of a non-terminating expression may be avoided. If a program terminates using both call-by-value and call-by name, then the same result will be returned in both cases, which is guaranteed by the *Church-Rosser Property*. Thus, the non-strict language avoids redundant evaluations and is more expressive than the strict language. That is, it increases the power of functional abstraction and allows the definition of infinite structures.

- *Implicit Storage Management :* Another major feature of functional languages is that there is no notion of *explicit* storage management. The programmer is relieved from having to think about how or where the data objects in a program are stored within the computer memory and from writing code to recover inaccessible memory and reuse it. This is an important abstraction from the housekeeping needed with common imperative languages. Data types can be defined at an abstract level with no concern as to how they are represented internally, this being handled automatically by implementation. Furthermore, the lifetime of the object is also unimportant as far as the programmer is concerned and so we do not have to worry about when an object ceases to be required by the program. Built-in operations on data automatically allocate storage as needed and storage that becomes inaccessible is then implicitly deallocated by triggering garbage collection. The absence of explicit code for managing storage effectively for data objects makes programs simpler and shorter.

- *Static (Implicit) Polymorphic Typing :* A type system for a language is a set of rules for associating a type with expressions in the language to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up. Most modern functional languages adopt a rich static strong polymorphic type system ([19], [27], [60], [69]) in which polymorphism is allowed in both primitive and user-defined functions, and a type inference system can be used to infer the types of expressions when little or no type information is given explicitly. Static strong typing, which means that expressions are type consistent and thus type errors can not occur at run-time, and that the type of every expression can be determined by static program analysis rather than during program execution, helps in debugging since one is guaranteed that if a program compiles successfully then no error can occur at run-time due to type violations. It also leads to more efficient implementations, since it allows one to eliminate most run-time tags and type testing, but it may lead to a loss of flexibility and expressive power. While a *monomorphic* type system in which every value can be interpreted to be of one and only one type is too restrictive

in its expressive power, a function that is assigned a type expression with one or several type variables is said to be *polymorphic*, which means that it can have several different types. Overloading can likewise assume different types. However, it is just the use of one function symbol for many different but usually related functions. For an overloaded function, a definition must be given for each possible argument type while a polymorphic function is defined in a single definition. Polymorphic functions are extremely useful. In one function definition, we define a number of functions that are computationally similar. Without polymorphism we have to give one definition for each case. Polymorphism makes programming simpler since we can define functions that are useful in a variety of applications. Polymorphism is not confined to functional languages, but it reflect the importance of higher-order functions to the expressive power of functional languages. It is generally desirable for a language to be statically checked using a powerful and strong type system.

1.2 Storage Management Overhead

A high implementation overhead is required to support a variety of features that are provided by a functional language. Despite a number of approaches to implementing functional languages, all implementations deal with the same basic underlying issues, i.e. dynamic allocation of heap objects and garbage-collected reclamation of them. The overhead due to storage management is one of the major overheads of functional language implementations. The principal sources of the storage management overhead are dynamic allocation and automatic reclamation of indefinite extent storage.

Dynamic Heap Allocation

Heap is a storage which is allocated and reclaimed dynamically in any order at any time during a program's execution. There are two kinds of run-time objects to be dynamically allocated in indefinite-extent heap storage during the execution of a functional program ([31]):

- Objects built explicitly by the program such as records, lists and trees whose size can vary during the running of a program and thus cannot be statically determined.
- Objects built by the implementation such as closures for representing function values and delayed expressions.

In the implementation of a language with implicit storage management, data structures which are defined explicitly in a program but whose size cannot be statically determined, such as lists, trees, records and so on, are generally allocated in the heap. Heap is also needed to support higher-order functions and non-strict/lazy semantics of functional languages. The need for heap allocation arises when parameters and locally defined objects within a function outlive a call to that function. In programming languages which do not support higher-order functions, lifetime of storage for local variables are confined to a function's activation record. Thus, storage for the locals is allocated when activation begins and is deallocated when the activation ends.

A closure is a means of representing function values, which consists of code together with its free variables. The use of closures guarantees that variables are statically scoped. In languages with higher-order functions and lexical scoping rules, it is possible to write a function, say f, which returns a lexically enclosed function, say g. Allocating the environment for the function value on the heap, however, adds a large cost to each function call.

To implement non-strict semantics and lazy evaluation, any implementation strategy uses a notion of *delayed* expression in some way, but in the abstract they all do the same thing: delay the evaluation of an expression until its value is demanded, then evaluate it whenever needed (in normal-order implementation), and save the value so that it may be used on future demands without recomputation (in lazy implementation). Delayed representation of an expression must contain enough information to enable the expression to be evaluated later. Moreover, in order to implement lazy evaluation in which the expression is guaranteed to be evaluated at most once, there must additionally be some mechanism to cache the value of the expression and return it later rather re-evaluating it. Creating a delayed expression requires the allocation of some storage. Because the value of the delayed expression might be needed at some unknown time in future, it must be allocated in a heap which has an indefinite extent.

Automatic Storage Reclamation

An implementation of a programming language with dynamic heap allocation requires some kind of storage reclamation mechanism, either explicit or implicit(automatic), because the heap storage occupied by objects which are inaccessible and are no longer required in the execution of the program have to be automatically reclaimed so that the physically finite storage is not rapidly exhausted by the program. Automatic storage reclamation is espe-

cially important in implementations of declarative languages, such as functional languages, that have no notion of explicit storage control and tend to use storage extensively. There are three basic approaches, with a number of variants, to automatically reuse the portions of heap storage, called garbage, that have previously been allocated but are no longer used by the program: ([24], [70]) *reference counting, mark-and-sweep garbage collection*, and *copying garbage collection.*

On most current computer systems, heap allocation and automatic storage reclamation is relatively expensive. A significant amount of overhead both in time and in space, are incurred in the process of automatic storage reclamation. A substantial portion of the execution time is spent in automatic storage reclamation. Due to this storage management overhead, in conventional computer architectures, typical implementations of functional programs are inefficient and waste memory, and thus functional programs tend to be much slower than their imperative equivalents.

1.3 The Role of Lifetime Information

Like many other programming language implementations, efficient storage management is a central concern in functional language implementations. The approaches to reduce the storage management overhead due to dynamic heap allocation and garbage collection in functional language implementations can generally be classified into three ways as follows:

1. Avoid heap allocation and garbage collection by using a storage structure (for example, a stack) whose allocation and reclamation is more efficient than for a heap. A heap provides a very general storage allocation mechanism, but it is also very expensive. In contrast, a stack is much less flexible allocation mechanism, but the store it allocates is recovered immediately when it becomes unused, and this recovery simply involves decrementing the stack pointer.

2. Even though heap allocation and garbage collection is used, reduce the frequency of invoking the garbage collection by reducing the frequency of running out of free storage.

3. Even though the garbage collection is invoked, improve the garbage collection scheme itself rather than fully relying on the original garbage collection scheme.

We describe a variety of storage management optimization techniques as follows:

- *Stack Allocation* : Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in the stack and cheaply reclaimed without invoking garbage collection. ([16], [20], [34], [42], [55], [76])

- *Explicit Reclamation and In-place Reuse* : When heap-allocated objects are no longer needed, they can be reclaimed into a free storage list explicitly by the program without invoking a garbage collection process. When heap-allocated objects are no longer needed, their storage can be reused directly in the allocation of new objects without invoking garbage collection. ([11], [35], [41], [49], [50], [51])

- *Reference Counting Elimination* : In reference counting, each object contains a count, called the reference count, of the number of references (pointers) pointing to it. Each time a reference is created or destroyed its reference count needs to be incremented or decremented. It is desirable for unnecessary updatings on reference counts at each reference's creation/ deletion to be avoided. ([9], [32])

- *Block Allocation/Reclamation* : A number of objects are allocated together in a contiguous block of a heap storage and the whole block is put on the free list, rather than the individual objects. This allows reclamation of larger segments of storage, and reduces run-time overhead by avoiding the traversal of the individual objects (in mark-sweep collection, for instance). ([76])

The main reason that objects are allocated on a heap is that their lifetime is generally unknown at compile-time and thus is assumed to have indefinite extent. In principle, garbage collection can be avoided by compile-time scheduling of storage use. Information about the lifetimes of objects can have an important role in storage management optimizations. Objects allocated on the heap could be allocated and reclaimed more efficiently if compilers sought to extract information about their lifetimes from the program text instead of making worst-case assumptions. Stack allocation can be safely applied when the lifetimes of objects are contained in the lifetime of a storage which can be reclaimed in the reverse order of that in which it is allocated. Explicit reclamation and in-place reuse can be applied when the lifetimes of heap-allocated objects are known. Reference counting elimination can be applied when the lifetimes of references to a heap-allocated object are known. Block allocation/reclamation can be applied when a set of objects whose lifetimes are same are to be allocated on a heap.

1.4 Lazy Evaluation Overhead and the Role of Order-of-Demand Information

Many modern functional languages adopt a non-strict semantics and use the lazy evaluation model. In a non-strict functional language, arguments in a function application are not evaluated unless and until their values are demanded. An optimized version of normal-order evaluation is *lazy* evaluation in which each argument is evaluated only when its value is first demanded, and the value is saved for later demands. The implementation of lazy evaluation, however, involves substantial overhead due to

1. The need for delaying the evaluation of arguments.

2. The need for checking their evaluation status (that is, whether they have already been evaluated) every time their values are demanded.

In a non-strict functional language that is implemented using lazy evaluation, exact information about which arguments to a function will be demanded, what the order of evaluation among the arguments of a function is, and what the evaluation status of an argument is when its value is demanded, cannot generally be decided at compile-time. If such information could be safely approximated at compile-time, however, a number of important optimizations could be performed. Devising such analyses and optimizations has been a major focus of functional language research over the past decade.

Information about which arguments to a function will definitely be demanded, called *strictness information*, is used to optimize lazy evaluation by converting lazy evaluation into applicative-order evaluation and thus reducing the overhead of lazy evaluation ([36], [31], [63], [66]). Information about the *order of evaluation* of the arguments to a function can be useful for a number of optimizations, including copy elimination ([11], [30], [35]) and process scheduling in a parallel system [13]. Information on the *status of evaluation* of arguments when they are demanded is useful for eliminating unnecessary checking [14] and for efficient storage management of delayed expressions (closures).

These information can be reformulated in terms of information about the order in which the values of bound variables are demanded, called *order-of-demand* information, as follows:

- *Status of Evaluation*: Given an occurrence x_i of a variable x in the body of a function f, if for each possible execution of the body of f there exists another occurrence x_j of x such that x_j is demanded before x_i, then we know that x must have been evaluated

by the time x_i is encountered. Thus no run-time test of x's status is required to obtain the value of x_i.

- *Order of Evaluation*: Given two parameters x and y of a function f, if for every occurrence y_i of y in the body of f there exists an occurrence x_j of x that is demanded before y_i, then we can conclude that x will always be evaluated before y.

- *Strictness*: Given a parameter x of a function f, if we can determine that for each possible (terminating) execution of the body of f some occurrence x_i of x was demanded, then we can determine that f is strict with respect to x

1.5 Semantics-based Analysis

Compiler optimizations should not affect the standard semantics of programs, but may affect the pragmatics of programs. Any compile-time program analysis for optimization is required to be semantically correct or safe with respect to the standard semantics. For functional programming languages that have precise, straightforward semantics, *denotational semantics* ([4], [74], [79]) and *abstract interpretation* ([2], [17], [25], [26], [46], [44], [63]) are particularly powerful tools for general and effective program analyses to infer certain properties of programs that may be needed for semantics-preserving optimizations.

Denotational Semantics

To perform meaning-preserving optimizations, a precise semantics is needed for any programming language. Because of their sound theoretical foundation, functional languages are particularly well suited for semantics-based analysis of programs. Denotational semantics is a formal way of describing the (standard or non-standard) meaning of a program in terms of mathematical semantic domains that properly capture our intuition about program behaviors and semantic functions, and is the most widely used tool for describing the formal semantics of functional programming languages.

Domain theory and denotational semantics were introduced to give meaning to syntactic expressions. Formally, we consider the meaning of an expression to be a value taken from some *domain* with well understood mathematical properties. To properly represent the results of all computations, a suitable domain must include elements representing incompletely evaluated objects, and thus represent approximations to completely evaluated objects. Therefore, we need an ordering establishing a partial order based on the *definedness* of elements and a least element on the domain D representing the completely undefined

object (e.g. non-termination). We also require that domains include the limits of infinite chains of approximating partial elements, i.e. for each increasing sequence in D, the least upper bound exists, making D a complete partial order*(cpo)*. A domain D is called *flat* when all elements apart from the bottom element are incomparable with each other. For functions defined only on values which do not include compound data types such as lists, this type of domain is sufficient for defining a consistent semantics. In a domain which is not flat, called a *non-flat* domain, there may exist an ordering among all elements. Computable functions between domains should preserve the information ordering structure and the limits, i.e. they are *monotonic* and *continuous*. Given domains with these requirements, we can construct domains such as the Cartesian product, the function space, the separate sum, the coalesced sum, the reflexive domain, the functional domain and the powerdomain. We can now interpret recursively defined functions as least fixed points and also compute them by iteration from the *fixed point theorem* [74]. If the domain D is finite or it has the property that all chains are of finite length, then the least fixpoint can be computed in a finite steps by iteration. The fixpoint theorem also forms the basis for a number of practical algorithms in abstract interpretation.

In denotational semantics, the semantics of a language is defined by semantic functions mapping syntax of the language to a suitable domain that captures certain computational behavior. Such semantic functions are defined so that the meaning of any composite syntactic structure is expressed in terms of the meanings of its immediate constituents. The standard or non-standard semantics of any functional programming language can be expressed in terms of a domain of objects and continuous functions defined on it.

Abstract Interpretation

Abstract interpretation is a computation over some abstraction or approximation of a semantic domain. It is a formal methodology that mathematically approximates uncomputable semantic properties and can be related directly back to the original denotational semantics. The correctness can be proved at an abstract level, independently of operational concerns. From a practical perspective, abstract interpretation also provides a convenient methodology for expressing compile-time analyses in a relatively language-independent manner. Abstract interpretation can be used as a general technique for deducing information about a program from its text, by executing an abstract version of the program with abstract data and then extracting desired information from the abstract results, instead of actually executing the original program and then extracting some information via a given

standard or non-standard semantics. Abstract interpretation has the advantages of handling inter-functional dependency, recursion, higher-order functions, and aggregate data structures. Most information the we want to know about the program is essentially undecidable. Therefore the issues of effective computability and semantic correctness or safety become crucial for program analysis through abstract interpretation.

Abstract interpretation in the *functional* idiom analyzes certain properties of an expression's evaluation by first defining an appropriate abstract domain which is usually much simpler than the exact standard or non-standard semantic domain, having the minimum structure required to encapsulate these properties, and defining an abstract version of each function occurring in the expression on the abstract domain. Then the abstract version of each function is applied to the abstractions of its arguments to give a result, also in the abstract domain, from which the required properties of the function's real application can be deduced. The safety criteria of an abstract interpretation is that the real result of the application of a function is in the set represented by the result of the corresponding application of the abstracted function. To be certain of being correct, we must consider every possible outcome of the computations represented by a given abstract value, only one of which will occur in any particular instance. Of course, by enriching the abstract domain sufficiently, we could represent any property completely and make precise predictions possible, but ultimately we would arrive back at the given original standard or non-standard domain itself and have to do the whole computation anyway. Normally, the applications in the abstract domain are sufficiently simple that they can be effectively performed at compile-time. Thus, optimizations relying on information from abstract interpretation can be performed at compile-time.

1.6 Overview of the Thesis

The main reason that objects are allocated on a heap is that their lifetime are generally unknown at compile-time. This property of object is called indefinite extent. Information about the lifetimes of objects can have an important role in storage management optimizations. Objects allocated on the heap could be allocated and reclaimed more efficiently if compilers sought to extract information about their lifetimes from the program text instead of making worst-case assumptions. In functional languages among many other languages, exact information about lifetimes of objects is generally not known at compile-time, but can be known only at run-time. Such information, if inferred at compile-time, allows a variety of optimizations that reduce the storage management overhead.

In this thesis, using formal denotational semantics and the abstract interpretation technique, we develop a set of compile-time semantic analyses for higher-order, polymorphic functional languages with non-flat domains, which provide safe information about the lifetime of dynamically-allocated objects during a program's execution. Based on the information that is inferred statically through compile-time analyses, we investigate a variety of storage management optimization techniques that reduce the storage management overhead in functional language implementations including bounded-extent storage allocation, explicit reclamation, in-place reuse, reference counting elimination, and block allocation/reclamation.

1.6.1 The Functional Language

We introduce a simple higher-order functional language which is representative of a class of modern functional programming languages and will be used throughout the thesis. It is a common observation that the syntax of modern high level functional programming languages are sugared versions of the lambda calculus ([8], [67]). We view our analysis as being performed at the high-level expression-oriented source code, not at low-level target code. This view is more portable, though the other view has the possible advantage being able to use more traditional compiler optimization techniques. Because the information is derived at the source level it is available during code generation regardless of the target language of a given virtual machine.

As a model language, we define a simple higher-order functional language whose syntax is based on the typed lambda calculus augmented with constants that include primitive functions. The language syntax is defined in Figure 1.1. To support first-class functions, all functions including primitive ones are curried and explicit lambda abstractions are allowed. Nested groups of equations are also allowed. We assume that all identifiers have unique names, i.e. that the program has been alpha-converted to ensure that all bound variables have been given unique names. Data structures more general than lists, such as trees, are not dealt with here, however, the methods for lists could be extended to handle general free data types as well. The model language is defined this way because any functional language is essentially a sugared version of the lambda calculus, and a number of semantic analysis which we will discuss in this thesis are applied to high-level source programs. We assume that the functional language adopts a strong *static* type system.

c	$\in$	Con	Set of Constants(including primitive functions)
			$= \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}, \texttt{nil}, \texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{null}\}$
x	$\in$	Id	Set of Identifiers
e	$\in$	Exp	Set of Expressions, defined by
			$e ::= c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
			$\mid e_1 e_2 \mid \texttt{lambda}(x).e \mid \texttt{letrec}\ x_1 = e_1;\ \ldots;\ x_n = e_n;\ \texttt{in}\ e$
pr	$\in$	$Program$	Set of Programs, defined by
			$pr ::= \texttt{letrec}\ x_1 = e_1;\ \ldots;\ x_n = e_n;\ \texttt{in}\ e$

Figure 1.1: The Syntax of Functional Language

Notational Conventions

Throughout this thesis, the following conventional notations are adopted;

1. Double bracket, $[\![\ ]\!]$, is used to surround syntactic objects,
2. Square bracket and map arrow, $[\ \mapsto\ ]$, are used for environment updates,
3. Angle bracket, $\langle\ \rangle$, is used for tupling,
4. Subscripts of (1), (2) and (3) are used to denote the first, second and third element of a tuple, respectively and a subscript of (1,2) is used to denote a pair consisting of the first and second elements of a tuple.

1.6.2 Strict and Non-strict Standard Semantics

In strict languages, the reduction of a function application is performed in applicative-order and thus the application of a function to its arguments results in the arguments being evaluated before they are passed to the function. This section describes the standard denotational semantics for the model higher-order functional language.

Standard Semantic Domains

The meaning of an expression under the standard semantics is the value of the expression that we usually think of, such as a number, boolean value, function or list. The standard semantic domain D_s and the domain of standard environments E_s that is a domain of functions mapping identifiers on to their standard meaning are defined as follows:

$$
\begin{aligned}
D_s &= \sum_{\tau} D_s^{\tau} \qquad \text{/* Standard semantic domain */} \\
E_s &= Id \rightarrow D_s \quad \text{/* Domain of standard environments */}
\end{aligned}
$$

The standard semantic domain D_s is a separated sum domain consisting of a subdomain for each type. The standard subdomain D_s^{τ} for expressions of type τ is defined as follows:

$$
\begin{aligned}
D_s^{int} &= \{\perp_{int}\} + \{\ldots, -1, 0, 1, \ldots\} && \text{subdomain for integers} \\
D_s^{bool} &= \{\perp_{bool}\} + \{true, false\} && \text{subdomain for booleans} \\
D_s^{\tau_1 \rightarrow \tau_2} &= D_s^{\tau_1} \rightarrow D_s^{\tau_2} && \text{subdomain for functions of type } \tau_1 \rightarrow \tau_2 \\
D_s^{\tau\ list} &= \{\perp_{\tau\ list}\} + \{nil^{\tau\ list}\} && \text{subdomain for lists of type } \tau\ list \\
&\quad +(D_s^{\tau} \times D_s^{\tau\ list})
\end{aligned}
$$

Standard Semantic Functions

We introduce the standard strict semantic functions as follows:

$$
\begin{aligned}
S_c &: Con \rightarrow D_s && \text{/* Standard semantic function for constants */} \\
S_e &: Exp \rightarrow E_s \rightarrow D_s && \text{/* Standard semantic function for expressions */} \\
S_{pr} &: Program \rightarrow D_s && \text{/* Standard semantic function for programs */}
\end{aligned}
$$

The standard semantic function S_c gives standard meaning to constants. The standard semantic function S_e gives standard meaning to expressions in a given standard environment for identifiers. The standard semantic function S_{pr} gives standard meaning to programs. The semantic equations for the standard semantic functions are expressed in Figure 1.2. Here, the symbol of λ_s is used to denote a strict function abstraction compared with an ordinary notation of λ.

$$
(\lambda_s x^{\tau_1}.e^{\tau_2})\ y = \begin{cases} \perp & y = \perp_{\tau_1} \\ (\lambda x.e)\ y & y \neq \perp_{\tau_1} \end{cases}
$$

env_s denotes any standard environment in E_s, and $nullenv_s$ is a standard environment that maps every identifier on to the least element of its standard semantic domain. Note that S_e and env_s' is recursively defined.

The standard semantic functions for non-strict semantics are defined as the standard semantic functions for strict semantics in which λ_s is replaced by λ.

1.6.3 Organizations

The rest of the thesis is organized as follows:

$$
\begin{array}{lcl}
S_c[\![c]\!] & = & c, c \in \{\ldots, \mathtt{-1}, \mathtt{0}, \mathtt{1}, \ldots\} \\
S_c[\![\mathtt{true}]\!] & = & \mathit{true} \\
S_c[\![\mathtt{false}]\!] & = & \mathit{false} \\
S_c[\![\mathtt{nil}^{\tau\ list}]\!] & = & \mathit{nil}^{\tau\ list} \\
S_c[\![\mathtt{cons}]\!] & = & \lambda_s x.\lambda_s y.\ \mathrm{pair}(x,y) \text{where } \mathrm{pair}(x,y) = \langle x, y\rangle \\
S_c[\![\mathtt{car}]\!] & = & \lambda_s x.\ \mathrm{first}(x) \text{where } \mathrm{first}(x) = x_{(1)} \\
S_c[\![\mathtt{cdr}]\!] & = & \lambda_s x.\ \mathrm{second}(x) \text{where } \mathrm{second}(x) = x_{(2)} \\
S_c[\![\mathtt{null}]\!] & = & \lambda_s x.\ \text{if } (x = \mathit{nil}) \text{ then } \mathit{true} \text{ else } \mathit{false}
\end{array}
$$

$$
\begin{array}{lcl}
S_e[\![c]\!]env_s & = & S_c[\![c]\!] \\
S_e[\![x]\!]env_s & = & env_s[\![x]\!] \\
S_e[\![e_1 + e_2]\!]env_s & = & S_e[\![e_1]\!]env_s + S_e[\![e_2]\!]env_s \\
S_e[\![e_1 - e_2]\!]env_s & = & S_e[\![e_1]\!]env_s - S_e[\![e_2]\!]env_s \\
S_e[\![e_1 = e_2]\!]env_s & = & \text{if } (S_e[\![e_1]\!]env_s = S_e[\![e_2]\!]env_s) \text{ then } \mathit{true} \text{ else } \mathit{false} \\
S_e[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]env_s & = & \text{if } (S_e[\![e_1]\!]env_s = \mathit{true}) \\
 & & \quad \text{then } S_e[\![e_2]\!]env_s \text{ else } S_e[\![e_3]\!]env_s \\
S_e[\![e_1 e_2]\!]env_s & = & (S_e[\![e_1]\!]env_s)(S_e[\![e_2]\!]env_s) \\
S_e[\![\mathtt{lambda}(x).e]\!]env_s & = & \lambda_s y.S_e[\![e]\!]env_s[x \mapsto y] \\
S_e[\![\mathtt{letrec}\ x_1 = e_1; \ldots; x_n = e_n; \mathtt{in}\ e]\!]env_s & = & S_e[\![e]\!]env'_s
\end{array}
$$

$$\text{where } env'_s = env_s[x_1 \mapsto S_e[\![e_1]\!]env'_s, \ldots, x_n \mapsto S_e[\![e_n]\!]env'_s]$$

$$S_{pr}[\![pr]\!] = S_e[\![pr]\!]nullenv_s$$

Figure 1.2: Standard Semantic Functions

- **Chapter 2. Escape Analysis** : In a higher-order functional language, exact information about the lifetime of objects, such as bound variables, within a function compared to the lifetime of the activation of the function call is generally unknown at compile-time, and can only be completely determined at run time. This chapter describes a method for computing, at compile-time, safe information about the relative lifetime of arguments and local objects within a function with respect to the lifetime of an activation of the function call. This analysis is described for a monomorphic, strict, higher-order functional language. The method is based on a compile-time semantic analysis called *escape analysis* which provides information about the lifetimes of objects within a function with respect to the lifetime of the activation of a function call. This information is called the escapement of objects.

- **Chapter 3. Refinements of Escape Analysis** : For structured objects such as lists and trees, the escape information that is obtainable through the escape analysis is rather coarse. This chapter describes a method for computing more refined escape information for a monomorphic, strict, higher-order functional language. This method is based on a compile-time semantic analysis called *refined escape analysis* which is an extension of escape analysis and determines at compile-time how much of an object outlives the activation of the function call in which it is created.

- **Chapter 4. Reference Escape Analysis** : In a higher-order functional language, exact information about the lifetime of a dynamically created reference (pointer) to a heap-allocated object is generally unknown at compile-time. This chapter describes a method for computing, at compile-time, safe information about the relative lifetime of dynamically created references. This method is based on a compile-time semantic analysis called *reference escape analysis*.

- **Chapter 5. Order-of-Demand Analysis** : In a non-strict functional language with lazy evaluation, exact information about the strictness of arguments, the order of evaluation among arguments, and the evaluation status of arguments when demanded is generally unknown at compile-time. This chapter describes a method for statically inferring a range of information including strictness, evaluation-order, and evaluation-status information. This method is based on a compile-time analysis called *order-of-demand analysis* which provides safe information about the order in which the values of bound variables are demanded.

- **Chapter 6. Polymorphic Invariance** : All the semantic analyses presented in the preceding chapters have dealt with a higher-order functional language with a *monomorphic* type system. Using the notion of *polymorphic invariance*, this chapter describes a method for applying escape analysis, reference escape analysis, and order-of-demand analysis to a polymorphic language using the analysis techniques for a monomorphic language.

- **Chapter 7. Extensions to Non-strict Languages** : Many modern functional languages adopt a non-strict semantics using the lazy evaluation model, which is more powerful than strict languages in its expressiveness. Using program transformation, this chapter describes the extensions of escape analysis and reference escape analysis to a non-strict language.

- **Chapter 8. Storage Management Optimizations** : The escape information that is inferred at compile-time from the semantic analyses which have been described in previous chapters, allows a variety of storage management optimizations in functional language implementations. Using the statically inferred escape information, this chapter describes a variety of optimization techniques to reduce the storage management overheads in functional language implementations, including stack(bounded-extent storage) allocation, explicit reclamation, in-place reuse of garbage cells, reference counting elimination, block allocation/reclamation, and improving generational garbage collection.

- **Chapter 9. Related Work, Conclusions, and Future Work** : This chapter surveys some previous work related to the work presented in this thesis, summarizes the contributions of this thesis, and suggest some further research in this area.

Chapter 2

Escape Analysis

In higher-order functional languages, exact information about the relative lifetime of an object with respect to the lifetime of the activation of the function call that creates the object is generally unknown at compile-time. When storage is allocated for such an object, it is generally allocated from a heap. The object is then reclaimed using some kind of automatic reclamation method. Lifetime information, if inferred at compile-time, can be useful for efficient management of storage for these objects at run-time.

In this chapter, we present a method for computing at compile-time safe information about the relative lifetime of arguments and local objects defined within a function with respect to the lifetime of an activation of the function call for a higher-order, monomorphic, strict functional language. This method is based on a compile-time semantic analysis called *escape analysis* which provides information about the lifetimes of such objects with respect to the activation of the function call that creates them. This property is called *escapement* of objects. First, using denotational semantics and abstract interpretation, we introduce a non-standard denotational semantics called *escape semantics* that describes the actual escape behavior, but is incomputable at compile time. An abstraction method for approximating the exact escape semantics which is both safe with respect to the exact escape semantics and computable at compile-time is then presented. Based on this *abstract escape semantics*, we describe the escape testing algorithms which determine escape information for functions that holds true for every possible application of the function, and also escapement information that holds for a particular call to the function. Another safe and computable abstraction of the exact escape semantics, called *improved abstract escape semantics*, that improves the precision of escape information that is obtainable through the abstract escape semantics is also presented using the position information of objects in a list structure. Finally, the

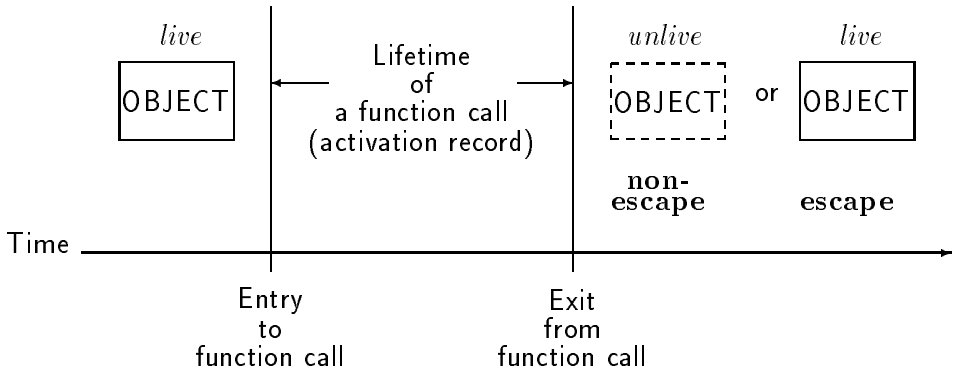

Figure 2.1: Escapement of Objects

complexity of the escape analysis is discussed.

2.1 Escapement of Objects

The lifetime of an object defined in a program is the period from the time it is created to the time it will be no longer used. The notion of relative lifetime is, given a pair of objects O_1 and O_2 being built during a program's execution, whether the object O_1 has a greater lifetime with respect to the lifetime of the other object O_2 or not. The notion of relative lifetime can be applied to any pair of objects that exist during a program's execution. We are particularly interested in the case when one object is either an argument to a function or a locally defined object, and the other object is an activation (record) of the function call, as is shown in Figure 2.1. That is, given a function, we are interested in whether an argument or a locally defined object escapes the activation of the function call or not. Note that the escaping argument or locally defined object needs to be retained after the activation to the function call ends. We formally define the notion of escapement of objects with respect to a function.

Definition 2.1 (Global/Local Escapement) Given a function f with n formal parameters and m locally defined objects, the i^{th} formal parameter or locally defined object is said to

- *escape* the function call to f *globally* if, in *some* possible application of f to n arguments, some or all of the corresponding actual parameter or local object outlives

the activation of the function call (by being contained in the result of the function application).

- *escape* the function call to f *locally* in $(f\ e_1\ \ldots e_n)$ if, in the particular function application of $(f\ e_1\ \ldots e_n)$, some or all of the corresponding actual parameter or local object outlives the activation of the particular function call to f (by being contained in the result of the function application of $(f\ e_1\ \ldots e_n)$).

The global escape information about a function can be safe in any context in which the function is called. In this sense, it is the property of function regardless of its application context. Thus, the global escape information is safe for a function in a particular context, but it may be weaker, in terms of its usefulness, than the escape information of a function in a particular context. The local escape information about a function is the property of the function in a particular context rather than the property of the function alone.

From the escape information about a parameter or local object with respect to a function, we can deduce information about its lifetime: If a parameter or local object does not escape the function call to f globally then we can conclude that the lifetime of the corresponding argument or local object that is created inside the function is confined within the lifetime of the function call in any possible application of f unless it is shared elsewhere. Similarly, if it does not escape the function call to f locally in a particular context of $(f\ e_1\ \ldots e_n)$ then we can conclude that the lifetime of the corresponding argument or local object that is created inside the function is confined within the lifetime of that particular function call unless it is shared elsewhere.

Such escape information is generally unknown at compile-time in higher-order functional languages. We will develop a method for monomorphic, higher-order, strict functional languages to answer the following questions at compile-time:

- Given a function, which parameter or local object that is defined inside the function escapes the function call globally ?

- Given a function in a particular application context, which parameter or local object that is defined inside the function escapes that particular function call locally ?

A naive approach would be through syntactic analysis. However, such syntax-based approach is not sufficient for specifying higher-order escapements of objects. We formalize the escape model through denotational semantics and then derive a safe, computable analysis using the abstract interpretation. First, we define a non-standard denotational semantics that captures the exact operational notion of escapements. Since the functional language

we are dealing with allows higher-order functions, the result of an expression may be a function. Such a function, represented by a closure, has two important characteristics with respect to the escape semantics:

1. The closure is an object itself. We may be interested in whether the closure escapes or not, or we may be interested in another object that is captured(bound) within the closure. Thus, the value of an expression returning a function must indicate whether an interesting object has escaped.

2. A function value may be applied to arguments (which may themselves escape from the application). Therefore, in the non-standard escape semantics, the escape value of a function must include its behavior as a function.

Thus, the value of an expression in the escape semantics is an element of a non-standard semantic domain and must have two components. First, it must contain information about what is contained within the value of the expression. Second, it must capture the functional behavior of the expression over the values in the escape semantic domain. We then define a suitable, i.e. safe and computable, abstraction of the exact non-standard semantics that can be computed at compile time but provide less precise information. Then we describe how the abstract escape semantics can be used to gain global and local escape information. We also describe another abstraction of the exact escape semantics that provides more precise escape information but at a higher cost.

2.2 Exact Non-standard Escape Semantics

We introduce an exact, but incomputable, *non-standard* denotational semantics called *escape semantics*, which exactly describes the actual operational notion of escapement for functions in a program. Since the exact escapement during a program's execution depends on the standard values themselves, for example, the standard value of the predicate part of the conditional will determine which alternative will be taken, any exact escape semantics needs to contain the information about the standard meaning as well as escape information.

Each parameter or local object of a function will be analyzed separately to determine its escape behavior. We say that a parameter is *interesting* if it is the one whose escape behavior we are trying to determine. Thus, our escape semantics is defined in terms of a single interesting object.

Figure 2.2: The Basic Escape Domain

Representing Escape Information

The first step in describing a non-standard escape semantics is to define a suitable non-standard escape semantic domain. A *domain* is used, rather than a *set*, to guarantee that a recursive function always has a solution. The meaning we will attach to the syntax is the information about containment of interesting objects. For each expression, its corresponding value in the escape semantic domain should be able to tell whether no part of an interesting object is contained in the value of the expression ("non-escape"), or whether some part of or all of an interesting object is contained in the value of the expression ("escape"). Thus, under our non-standard escape semantics, we represent the meaning of an expression as a pair, called *an escape pair* (in the style of [43]),

1. whose first element denotes the containment of an interesting object in the value of the expression, and

2. whose second element denotes the functional behavior of the expression defined over the escape pairs when the expression itself is applied to another expression.

For a non-list type expression, the corresponding value in the non-standard escape semantic domain D_o has two components; The first component is an element of a domain called a *basic escape domain*, B_o which is a two-element domain of 0 and 1 ordered by $0 \sqsubseteq 1$ as shown in Figure 2.2. The interpretation of elements of B_o is defined as follows:

- 1 : Some part of or all of an interesting object *is* contained in the value of the expression.

- 0 : *No* part of any interesting object is contained in the value of the expression.

The second component is a function over D_o, whose meaning is the functional behavior of the expression defined over the escape values when the expression e is applied to another

expression. For expressions which have no higher-order behavior, such as non-function type expressions, err, denoting a function that can never be applied, is used. For a list type expression, the corresponding value in the escape semantic domain is a list of the corresponding values of its components in the escape semantic domain.

Escape Semantic Domains

D_o, the escape semantic domain, and E_o, the domain of environments mapping identifiers to their escape meanings, are completely defined as follows:

$$\begin{aligned} D_o &= \sum_{\tau} D_o^{\tau} \qquad \text{/* Escape semantic domain */} \\ E_o &= Id \rightarrow D_o \quad \text{/* Domain of escape environments */} \end{aligned}$$

The escape semantic domain D_o is a separated sum domain made up of the subdomains for each type defined by D_o^{τ}. The escape subdomain D_o^{τ} for expressions of type τ is defined as follows (in the style of [18]):

$$\begin{array}{lcll} D_o^{int} & = & B_o \times \{err\} & \text{subdomain for integers} \\ D_o^{bool} & = & B_o \times \{err\} & \text{subdomain for booleans} \\ D_o^{\tau_1 \rightarrow \tau_2} & = & B_o \times (D_o^{\tau_1} \rightarrow D_o^{\tau_2}) & \text{subdomain for functions of type } \tau_1 \rightarrow \tau_2 \\ D_o^{\tau\ list} & = & (B_o \times \{err\}) + (D_o^{\tau} \times D_o^{\tau\ list}) & \text{subdomain for lists of type } \tau\ list \end{array}$$

The escape subdomains for integers and boolean values is the cartesian product of B_o and err, which is ordered as follows:

$$\forall u, v \in D_o^{int\ or\ bool},\ \ u \sqsubseteq v \text{ iff } (u_{(1)} \sqsubseteq v_{(1)}) \text{ and } (u_{(2)} \sqsubseteq v_{(2)})$$

The escape subdomain for function of type $\tau_1 \rightarrow \tau_2$ is the cartesian product of B_o and the function space $D_o^{\tau_1} \rightarrow D_o^{\tau_2}$. The function space of $D_o^{\tau_1} \rightarrow D_o^{\tau_2}$ is ordered as follows:

$$\forall f, g \in D_o^{\tau_1} \rightarrow D_o^{\tau_2},\ \ f \sqsubseteq g \text{ iff } \forall d \in D_o^{\tau_1}, f_{(2)}\ d \sqsubseteq g_{(2)}\ d$$

The escape subdomain for lists of type $\tau\ list$ is the sum domain of the cartesian product of B_o and err, and the cartesian product of D_o^{τ} and $D_o^{\tau\ list}$, whose ordering is defined as follows:

$$\forall u, v \in D_o^{\tau\ list},\ \ u \sqsubseteq v \text{ iff } (\bigsqcup_{p\ \mathbf{in}\ u} p) \sqsubseteq (\bigsqcup_{q\ \mathbf{in}\ v} q)$$

where p **in** u denotes that p is an escape pair in u.

The bottom elements $\perp_{int}$ and $\perp_{bool}$ in the escape subdomains for integers and booleans are $\langle 0, err \rangle$, respectively. The bottom element $\perp_{\tau_1 \rightarrow \tau_2}$ in the escape subdomain for functions

of type $\tau_1 \rightarrow \tau_2$ is $\langle 0, \lambda x.\bot_{\tau_2} \rangle$. The bottom element $\bot_{\tau\ list}$ in the escape subdomain for lists of type $\tau\ list$ is $\langle 0, err \rangle$.

The top elements $\top_{int}$ and $\top_{bool}$ in the escape subdomains for integers and booleans are $\langle 1, err \rangle$, respectively. The top element $\top_{\tau_1 \rightarrow \tau_2}$ in the escape subdomain for functions of type $\tau_1 \rightarrow \tau_2$ is $\langle 1, \lambda x.\top_{\tau_2} \rangle$. The top element $\top_{\tau\ list}$ in the escape subdomain for lists of type $\tau\ list$, however, does not exist.

Escape Semantic Functions

We now introduce the non-standard escape semantic functions to give the syntax the escape meaning as follows:

$$
\begin{array}{lcll}
O_c & : & Con \rightarrow D_o & /* \text{ Escape semantic function for constants } */ \\
O_e & : & Exp \rightarrow E_o \rightarrow D_o & /* \text{ Escape semantic function for expressions } */ \\
O_{pr} & : & Program \rightarrow D_o & /* \text{ Escape semantic function for programs } */
\end{array}
$$

The escape semantic function O_c gives the non-standard escape meaning to constants. The escape semantic function O_e gives the non-standard escape meaning to expressions in a given escape environment for identifiers. The escape semantic function O_{pr} gives the non-standard escape meaning to programs The semantic equations for the escape semantic functions are expressed in Figure 2.3.

Since an interesting object is definitely not contained in constants such as integers, booleans and nil, and also such constants can never be applied, their values in escape semantics are given by $\langle 0, err \rangle$. The value of `cons` under escape semantics, when applied to two arguments, simply returns a pair consisting of the values of the two arguments. Since `cons` is curried, the result of applying `cons` to a single argument returns a closure containing that argument. The values of `car` and `cdr` under escape semantics, when applied to a list whose value is a pair in $D_o^{\tau\ list}$, returns the first and second components of the pair, respectively. The value of `null` under the escape semantics, when applied to a list, returns the escape values of constant boolean values, namely $\langle 0, err \rangle$.

env_o is any exact escape environment in E_o. In order to return the actual escape value of each expression, we must be able to determine which branch of the conditional primitive `if` would be evaluated at run-time. Here, for convenience, we instead resort to an oracle called **Oracle** to choose the appropriate branch of the `if`. Under the escape semantics, the value of a lambda expression which denotes a function reflects whether an interesting object is contained in the resulting closure it as a free identifier, as well as its functional behavior when it is applied. Note that free identifiers are treated separately according to

whether they are of a list type or a non-list type. $nullenv_o$ is an escape environment that maps every identifier to the least element of its escape semantic domain.

Uncomputable at compile-time

Oracle, which is of type $Exp \rightarrow \{true, false\}$, is used to resolve the exact escape behavior of the conditional expression `if`. Since the branch of the `if` that is actually taken depends on the standard value of its predicate expression, this oracle must rely on the standard semantics somehow. One way of achieving this is by having the exact escape semantics directly compute the standard meanings as well as the escape meanings of expressions, i.e. operate on elements in the domain $D_s \times D_o$. Thus, any exact escape semantics should contain all the standard meanings and all the escape meanings. Since the interpretation of programs under the standard semantics is uncomputable, interpretation under the exact non-standard escape semantics is not computable at compile time. In fact, from the point of view of information contents, the standard semantics can be considered as an abstraction of the exact escape semantics.

2.3 Abstract Escape Semantics

The escape semantics presented in the last section specifies exact escape information about functions in a program. But it is not suitable as a basis for compile-time analysis for two reasons:

1. Since conditionals cannot be evaluated at compile-time and there is no such thing as an oracle at compile-time, there is no way to know which branch of a conditional `if` will be executed.

2. We cannot know at compile time exactly which and how many elements each list will contain. When some part of a list is taken by either `car` or `cdr`, there is no way to know exactly which value is removed from the list and which values remain in the list.

For use by a compiler, we need a suitable escape semantics that will guarantee termination and yet still provide useful and safe information with respect to the exact escape semantics. To get some computable escape semantics, we need to somehow abstract or approximate the exact escape semantics. In general, abstraction or approximation of an exact standard or non-standard semantics can be done by abstracting either its domains or its primitive functions or both. Generally, there can be a range of possible computable abstractions, all of which are safe but which may vary in their information content and their complexities.

$$
\begin{aligned}
O_c[\![c]\!] &= \langle 0, err\rangle,\ \ c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}, \texttt{nil}^{\tau\ list}\} \\
O_c[\![\texttt{cons}]\!] &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\ \mathrm{pair}(x, y)\rangle\rangle \\
O_c[\![\texttt{car}]\!] &= \langle 0, \lambda x.\ \mathrm{first}(x)\rangle \\
O_c[\![\texttt{cdr}]\!] &= \langle 0, \lambda x.\ \mathrm{second}(x)\rangle \\
O_c[\![\texttt{null}]\!] &= \langle 0, \lambda x.\langle 0, err\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
O_e[\![c]\!]env_o &= O_c[\![c]\!] \\
O_e[\![x]\!]env_o &= env_o[\![x]\!] \\
O_e[\![e_1 + e_2]\!]env_o &= \langle 0, err\rangle\ \text{/* same for } e_1 - e_2 \text{ and } e_1 = e_2\ \text{*/} \\
O_e[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]env_o &= \text{if } \mathbf{Oracle}(e_1) \text{ then } O_e[\![e_2]\!]env_o \\
&\qquad\qquad \text{else } O_e[\![e_3]\!]env_o \\
O_e[\![e_1 e_2]\!]env_o &= (O_e[\![e_1]\!]env_o)_{(2)}\ (O_e[\![e_2]\!]env_o) \\
O_e[\![\texttt{lambda}(x).e]\!]env_o &= \langle V, \lambda y.O_e[\![e]\!]env_o[x \mapsto y]\rangle
\end{aligned}
$$

where

$V = 0 \sqcup (\bigsqcup_{z \in F^{non-list}} (env_o[\![z]\!])_{(1)}) \sqcup (\bigsqcup_{z \in F^{list}} (\bigsqcup_{p\ \mathbf{in}\ (env_o[\![z]\!])} p_{(1)}))$,

$p\ \mathbf{in}\ (env_o[\![z]\!])$ denotes that p is an escape pair in $env_o[\![z]\!]$,

$F^{non-list}$ = Set of non-list type free identifiers in $(\texttt{lambda}(x).e)$, and

F^{list} = Set of list type free identifiers in $(\texttt{lambda}(x).e)$.

$$
O_e[\![\texttt{letrec}\ x_1 = e_1; \ldots; x_n = e_n; \texttt{in}\ e]\!]env_o = O_e[\![e]\!]env'_o
$$

where $env'_o = env_o[x_1 \mapsto O_e[\![e_1]\!]env'_o, \ldots, x_n \mapsto O_e[\![e_n]\!]env'_o]$

$$
O_{pr}[\![pr]\!] = O_e[\![pr]\!]nullenv_o
$$

Figure 2.3: Escape Semantic Functions

Figure 2.4: The Abstract Basic Escape Domain

Abstracting Escape Semantic Domains

We present a suitable, i.e. safe and computable but less complete, abstraction of the exact escape semantics defined in the last section that allows an approximation of the exact escape behavior to be found at compile-time.

We safely approximate the exact escape semantics by abstracting escape semantic subdomains for list type expressions and by approximating escape semantic functions. For each expression, its corresponding value in the abstract escape semantic domain tells that no part of an interesting object is contained in the value of the expression ("non-escape"), or that some part of an interesting object may be contained in the value of the expression ("possible escape"). The abstract basic escape domain $\hat{B}_o$ is a two-element domain of 0 and 1 ordered by $0 \sqsubseteq 1$, and is similar to the basic escape domain B_o as shown in Figure 2.4. But, the interpretation of elements of $\hat{B}_o$ is defined differently from the interpretation of elements in B_o as follows:

- 1 : Some part of or all of an interesting object *may be* contained in the value of the expression.
- 0 : *No* part of any interesting object is contained in the value of the expression.

Note that the most precise information appears at the bottom of the domain and the least precise at top of the domain. The 0 and 1 may be thought of as defining sets of possible values of the expressions, and the domain is ordered by the subset ordering. Abstraction of the escape semantic subdomains for list type expressions is done by representing lists as finite objects, i.e. by combining the escape pairs of all its elements into a single escape pair.

The abstract escape semantic domain $\hat{D}_o$ is an abstraction of D_o, and the domain $\hat{E}_o$ is abstract escape environments $\hat{E}_o$. They are defined as follows:

$$\hat{D}_o = \sum_{\tau} \hat{D}_o^{\tau} \quad \text{/* Abstract escape semantic domain */}$$
$$\hat{E}_o = Id \to \hat{D}_o \quad \text{/* Domain of abstract escape environments */}$$

The abstract escape subdomain, $\hat{D}_o^{\tau}$ for expressions of type τ is defined as follows:

$\hat{D}_o^{int}$	$=$	$\hat{B}_o \times \{err\}$	abstract subdomain for integers
$\hat{D}_o^{bool}$	$=$	$\hat{B}_o \times \{err\}$	abstract subdomain for booleans
$\hat{D}_o^{\tau_1 \to \tau_2}$	$=$	$\hat{B}_o \times (\hat{D}_o^{\tau_1} \to \hat{D}_o^{\tau_2})$	abstract subdomain for functions of type $\tau_1 \to \tau_2$
$\hat{D}_o^{\tau\ list}$	$=$	$\hat{D}_o^{\tau}$	abstract subdomain for lists of type $\tau\ list$

Note that the abstract escape subdomain $\hat{D}_o^{\tau\ list}$ is the same as the subdomain $\hat{D}_o^{\tau}$.

Abstracting Escape Semantic Functions

We now introduce an abstract escape semantic functions to give the syntax the escape meaning as follows:

$$\hat{O}_c : Con \to \hat{D}_o \quad \text{/* Abstract escape semantic function for constants */}$$
$$\hat{O}_e : Exp \to \hat{E}_o \to \hat{D}_o \quad \text{/* Abstract escape semantic function for expressions */}$$
$$\hat{O}_{pr} : Program \to \hat{D}_o \quad \text{/* Abstract escape semantic function for programs */}$$

As an abstraction of the exact escape semantic functions, the abstract escape semantic functions are given in Figure 2.5.

The abstract value of `cons` returns a single escape pair that is approximating a list of escape pairs. The abstract values of `car` and `cdr` just returns their arguments, respectively. The abstraction for the conditional expression `if` no longer makes an appeal to the **Oracle**, but rather takes the least upper bound of the escape values of both branches. Notice that the definition of the abstract escape semantic function $\hat{O}_e$ on the expression of `lambda`$(x).e$ is considerably simpler than the exact escape semantic function O_e. $\hat{env}_o$ is any abstract escape environment in $\hat{E}_o$, and the $\hat{nullenv}_o$ is an abstract escape environment that maps every identifier on to the least element of its abstract escape semantic domain.

Safety

The actual escape property of the program that is being approximated must imply the escape information gathered from the abstract escape semantics. The safety of the abstract escape semantics means that the interpretation under this abstract semantics will never produce wrong escape information with respect to what is obtained by the interpretation under the exact escape semantics.

$$
\begin{array}{lcl}
\hat{O}_c[\![c]\!] & = & \langle 0, err\rangle,\ \ c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\} \\
\hat{O}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (The bottom element in } \hat{D}_o^\tau \text{)} \\
\hat{O}_c[\![\texttt{cons}]\!] & = & \langle 0, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\
\hat{O}_c[\![\texttt{car}]\!] & = & \langle 0, \lambda x.x\rangle \\
\hat{O}_c[\![\texttt{cdr}]\!] & = & \langle 0, \lambda x.x\rangle \\
\hat{O}_c[\![\texttt{null}]\!] & = & \langle 0, \lambda x.\langle 0, err\rangle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\hat{O}_e[\![c]\!]e\hat{n}v_o & = & \hat{O}_c[\![c]\!] \\
\hat{O}_e[\![x]\!]e\hat{n}v_o & = & e\hat{n}v_o[\![x]\!] \\
\hat{O}_e[\![e_1 + e_2]\!]e\hat{n}v_o & = & \langle 0, err\rangle \text{ /* same for } e_1 - e_2 \text{ and } e_1 = e_2 \text{ */} \\
\hat{O}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\hat{n}v_o & = & (\hat{O}_e[\![e_2]\!]e\hat{n}v_o) \sqcup (\hat{O}_e[\![e_3]\!]e\hat{n}v_o) \\
\hat{O}_e[\![e_1e_2]\!]e\hat{n}v_o & = & (\hat{O}_e[\![e_1]\!]e\hat{n}v_o)_{(2)}\ (\hat{O}_e[\![e_2]\!]e\hat{n}v_o) \\
\hat{O}_e[\![\texttt{lambda}(x).e]\!]e\hat{n}v_o & = & \langle \hat{V}, \lambda y.\hat{O}_e[\![e]\!]e\hat{n}v_o[x \mapsto y]\rangle
\end{array}
$$

where

$\hat{V} = 0 \sqcup (\bigsqcup_{z \in F} (env_o[\![z]\!])_{(1)})$ and

F = Set of all free identifiers in $(\texttt{lambda}(x).e)$.

$$
\hat{O}_e[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n; \texttt{in } e]\!]e\hat{n}v_o \ = \ \hat{O}_e[\![e]\!]e\hat{n}v_o'
$$

where $e\hat{n}v_o' = e\hat{n}v_o[x_1 \mapsto \hat{O}_e[\![e_1]\!]e\hat{n}v_o', \ldots, x_n \mapsto \hat{O}_e[\![e_n]\!]e\hat{n}v_o']$

$$
\hat{O}_{pr}[\![pr]\!] \ = \ \hat{O}_e[\![pr]\!]nul\hat{l}env_o
$$

Figure 2.5: Abstract Escape Semantic Functions

Definition 2.2 (Non-standard Application) Let NAP_n be an apply operator for elements in the non-standard semantic domains D_o and $\hat{D}_o$, defined as follows: For $ep, ep_1, \ldots, ep_n \in D_o$ and $\hat{D}_o$,

$$\mathrm{NAP}_n(ep, ep_1, \ldots, ep_n) \stackrel{\mathrm{def}}{=} \begin{cases} ep & n = 0 \\ \mathrm{NAP}_{n-1}(ep_{(2)}ep_1, ep_2, \ldots, ep_n) & n > 0 \end{cases}$$

We introduce the notion of *safety* with respect to escape information which relates the exact escape semantics to the abstract escape semantics. Let u and v be values of an expression e of type τ or τ *list* in the abstract escape domain $\hat{D}_o$ and the exact escape domain D_o, respectively. Let n be the number of arguments that type τ can take before returning a value of a non-function type. We say that the abstract escape semantic value u is a *safe* approximation (with respect to exact escape information) of the exact escape semantic value v iff

$$\Big(\bigsqcup_{p \textbf{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k)} p_{(1)}\Big) \sqsubseteq \mathrm{NAP}_k(u, s_1, \ldots, s_k)_{(1)}$$

for all $k \leq n$ where s_i is a *safe* approximation of t_i for all $i \leq k$.

Theorem 2.1 (Safety) *For any expression e, and environments env_o and $e\hat{n}v_o$ such that for all y, $e\hat{n}v_o[\![y]\!]$ is a safe approximation of $env_o[\![y]\!]$, $\hat{O}_e[\![e]\!]e\hat{n}v_o$ is a safe approximation of $O_e[\![e]\!]env_o$. Thus, the escape information obtained by the exact escape semantics implies the escape information obtained by the abstract escape semantics.*

Proof : We can prove by structural induction on expression e.

I. Base Case:

1. $e = c$: $\hat{O}_e[\![c]\!]e\hat{n}v_o = \hat{O}_c[\![c]\!]$ and $O_e[\![c]\!]env_o = O_c[\![c]\!]$. For $c \in \{\ldots$,`-1`,`0`,`1`,$\ldots$, `true`, `false`, `nil`, `null` $\}$, $O_c[\![c]\!] = \hat{O}_c[\![c]\!] = \langle 0, err\rangle$. For $c =$ `cons`, it holds because $x_1 \sqcup x_2$ is safe for a list consisting of y_1 and y_2 if x_1 and x_2 are safe for y_1 and y_2, respectively. For $c \in \{$`car`, `cdr` $\}$, it holds because, if x is safe for y, x is clearly safe for both first(y) and second(y).

2. $e = x$: $\hat{O}_e[\![x]\!]e\hat{n}v_o = e\hat{n}v_o[\![x]\!]$ and $O_e[\![x]\!]env_o = env_o[\![x]\!]$. Since, for all y, $e\hat{n}v_o[\![y]\!]$ is safe for $env_o[\![y]\!]$, it clearly holds.

II. Structural Induction Step: Assume that $\hat{O}_e[\![e]\!]e\hat{n}v_o$ is safe for $O_e[\![e]\!]env_o$ for expressions such as e_0, e_1, e_2, e_3 and e_n (structural induction hypothesis). Then, we show that $\hat{O}_e[\![e]\!]e\hat{n}v_o$ is safe for $O_e[\![e]\!]env_o$ for $e = e_1 + e_2$, `if` e_1 `then` e_2 `else` e_3, e_1e_2, `lambda`$(x).e_1$, and `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0 as follows:

1. $e = e_1 + e_2$: $\hat{O}_e[\![e_1 + e_2]\!]e\hat{n}v_o = O_e[\![e_1 + e_2]\!]env_o = \langle 0, err\rangle$. Similarly, it holds for $e_1 - e_2$ and $e_1 = e_2$.

2. $e =$ `if` e_1 `then` e_2 `else` e_3: $\hat{O}_e[\![$`if` e_1 `then` e_2 `else` $e_3]\!]e\hat{n}v_o = \hat{O}_e[\![e_2]\!]e\hat{n}v_o \sqcup \hat{O}_e[\![e_2]\!]e\hat{n}v_o$. $O_e[\![$`if` e_1 `then` e_2 `else` $e_3]\!]env_o$ is either $O_e[\![e_2]\!]env_o$ or $O_e[\![e_2]\!]env_o$ depending on the standard semantic value of e_1. In either case, by the structural induction hypothesis, $\hat{O}_e[\![e]\!]e\hat{n}v_o = O_e[\![e]\!]env_o$.

3. $e = e_1e_2$: $\hat{O}_e[\![e_1e_2]\!]e\hat{n}v_o = (\hat{O}_e[\![e_1]\!]e\hat{n}v_o)_{(2)}\ (\hat{O}_e[\![e_2]\!]e\hat{n}v_o)$. $O_e[\![e_1e_2]\!]env_o = (O_e[\![e_1]\!]env_o)_{(2)}$ $(O_e[\![e_2]\!]env_o)$. By the structural induction hypothesis, $(\hat{O}_e[\![e_1]\!]e\hat{n}v_o)$ and $(\hat{O}_e[\![e_2]\!]e\hat{n}v_o)$ are safe for $(O_e[\![e_1]\!]env_o)$ and $(O_e[\![e_2]\!]env_o)$, respectively. Then, by the definition of safe, it holds.

4. $e =$ `lambda`$(x).e_1$: $\hat{O}_e[\![$`lambda`$(x).e]\!]e\hat{n}v_o = \langle \hat{V}, \lambda y.\hat{O}_e[\![e]\!]e\hat{n}v_o[x \mapsto y]\rangle$. $O_e[\![$`lambda`$(x).e]\!]$ $env_o = \langle V, \lambda y.O_e[\![e]\!]env_o[x \mapsto y]\rangle$. Since, for all y, $e\hat{n}v_o[\![y]\!]$ is safe for $env_o[\![y]\!]$, we have that $V \sqsubseteq \hat{V}$. By the structural induction hypothesis, $\hat{O}_e[\![e]\!]e\hat{n}v_o[x \mapsto y]$ is safe for $O_e[\![e]\!]env_o[x \mapsto y]$.

5. $e =$ `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0: $\hat{O}_e[\![$`letrec` $x_1 = e_1; \ldots;\ x_n = e_n;$ `in` $e_0]\!]e\hat{n}v_o = \hat{O}_e[\![e_0]\!]e\hat{n}v'_o$ where $e\hat{n}v'_o = e\hat{n}v_o[x_i \mapsto \hat{O}_e[\![e_i]\!]e\hat{n}v'_o]$. $O_e[\![$`letrec` $x_1 = e_1; \ldots;\ x_n = e_n;$ `in` $e_0]\!]$ $env_o = O_e[\![e_0]\!]env'_o$ where $env'_o = env_o[x_i \mapsto O_e[\![e_i]\!]env'_o]$. Here, $e\hat{n}v'$ and env' are recursively defined. We prove that $e\hat{n}v'_o$ is safe for env'_o for all y by fixpoint induction on the environments $e\hat{n}v'_o$ and env'_o as follows:

   1. Base Case: The first approximation $env_o^{'(0)}$ of $e\hat{n}v'_o$ is $e\hat{n}v_o[x_i \mapsto \perp]$. The first approximation $env_o^{'(0)}$ of env'_o is $env_o[x_i \mapsto \perp]$. Thus, for all y, $e\hat{n}v_o^{'(0)}[\![y]\!]$ is safe for $env_o^{'(0)}[\![y]\!]$. Then, by the structural induction hypothesis, $\hat{O}_e[\![e_0]\!]e\hat{n}v'_o$. is safe for $O_e[\![e_0]\!]env'_o$.

   2. Fixpoint Induction Step: Assume that, for some fixed $k \geq 0$, the k^{th} approximation $e\hat{n}v_o^{'(k)}[\![y]\!]$ is safe for $env_o^{'(k)}[\![y]\!]$ for all y. (fixpoint induction hypothesis) Then, The $(k+1)^{th}$ approximation $e\hat{n}v_o^{'(k+1)}$ is $e\hat{n}v_o[x_i \mapsto \hat{O}_e[\![e_i]\!]e\hat{n}v_o^{'(k)}]$, and $env_o^{'(k+1)}$ is $env_o[x_i \mapsto O_e[\![e_i]\!]env_o^{'(k)}]$. By the structural induction hypothesis, $\hat{O}_e[\![e_i]\!]e\hat{n}v_o^{'(k)}$ is safe for $O_e[\![e_i]\!]env_o^{'(k)}$. Thus, for all y, $e\hat{n}v_o^{'(k+1)}[\![y]\!]$ is safe for $env_o^{'(k+1)}[\![y]\!]$. Then, by the structural induction hypothesis, $\hat{O}_e[\![e_0]\!]e\hat{n}v'_o$ is safe for $O_e[\![e_0]\!]env'_o$.

□

Termination

A compiler should terminate on every valid program. In order for any compiler which uses an analysis based on the abstract escape semantics to be effective, the interpretation under the abstract escape semantics must also be effective. The effectiveness of interpretation under

the abstract escape semantics means that it terminates on every valid program. When the abstract version of a function derived by the abstract escape semantics is non-recursive, termination is guaranteed. Recursive functions, however, derive recursively defined functions as their abstract versions. From the fixed point theorem of domain theory, the recursive function is given by the least fixed point of the corresponding higher-order functional. That is, for the function

$$f = F(f)$$

where f is of type τ and F is a functional corresponding to the body of f, the meaning of f is defined to be the least function satisfying the above equation and the least fixpoint f can be found as follows:

$$f = \bigsqcup_{i \to \infty} F^i(\bot_\tau)$$

where $F^0(x) = x$ and $F^i(x) = F(F^{i-1}(x))$ and $\bot_\tau$ is the bottom element of the domain of type τ, i.e. $\hat{D}^\tau$.

Theorem 2.2 (Termination) *For any (finite) program pr $\in$ Program, $\hat{O}_{pr}[\![pr]\!]$ is computable, i.e. always terminates in finite number of steps. Thus, interpretation of pr under the abstract escape semantics terminates.*

Proof : We can prove that the interpretation of a program under the abstract escape semantics will terminate by showing that the least fixed point of any functional in the abstract escape semantic domain can be computed in a finite number of steps. That is, for all functions defined according to the above equation, there must exist some j such that

$$F^k(\bot_\tau) = F^j(\bot_\tau)$$

for all $k > j$. A way of showing that there exists such a j is to show that every functional F must be *monotonic* and that the fixpoint iteration is performed over a *finite* domain. First, every functional over escape pairs defined in the abstract escape semantics is composed of monotonic operators and the least upper bound operator $\sqcup$, which is monotonic. Thus, every functional in the abstract escape semantics is monotonic. Second, $\hat{D}_o^b$ is finite for each base type b. $\hat{D}_o^{\tau_1 \to \tau_2}$ is also finite whenever $\hat{D}_o^{\tau_1}$ and $\hat{D}_o^{\tau_2}$ are finite, each domain $\hat{D}_o^\tau$ for each type τ is finite. By induction, then, in a strongly typed system where all types are finite, all $\hat{D}^\tau$ are finite. When finding the least fixpoint of a function of type τ, we need only search over the subdomain $\hat{D}_o^\tau$ of $\hat{D}_o$ and $\hat{D}_o^\tau$ is finite. Thus the least fixpoint can

be computed in a finite number of steps. Hence any analysis based on the abstract escape semantics is guaranteed to terminate at compile-time. □

For example, consider the following function `f` of type $int \to int \to int$ defined by

```
f x y = if (x=0) then y else f (x-1) (x+y)
```

The corresponding function f under the abstract escape semantics is described by

$$\begin{aligned} f &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\hat{O}_e[\![\texttt{if (x=0) then y else f (x-1) (x+y)}]\!][\texttt{x} \mapsto x, \texttt{y} \mapsto y]\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup (f_{(2)}\langle 0, err\rangle)_{(2)}\langle 0, err\rangle\rangle\rangle. \end{aligned}$$

Since f is recursive, f is the least fixpoint of the functional F of type $(int \to int \to int) \to int \to int \to int$ defined by

$$F = \lambda f.\langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup (f_{(2)}\langle 0, err\rangle)_{(2)}\langle 0, err\rangle\rangle\rangle.$$

Then, the least fixpoint is found by the following fixpoint iteration:

$$\begin{aligned} f^{(0)} &= F(\bot_{int\to int\to int}) \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup (\bot_{int\to int\to int(2)}\langle 0, err\rangle)_{(2)}\langle 0, err\rangle\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup \bot_{int\to int(2)}\langle 0, err\rangle\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup \bot_{int}\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y\rangle\rangle \end{aligned}$$

$$\begin{aligned} f^{(1)} &= F(f^{(0)}) \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup ((\langle 0, \lambda x.\langle x_{(1)}, \lambda y.y\rangle\rangle)_{(2)}\langle 0, err\rangle)_{(2)}\langle 0, err\rangle\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup ((\langle 0, \lambda y.y\rangle)_{(2)}\langle 0, err\rangle\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup \langle 0, err\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y\rangle\rangle \end{aligned}$$

Since $f^{(0)} = f^{(1)}$, a fixpoint has been found. Thus, $f = \langle 0, \lambda x.\langle x_{(1)}, \lambda y.y\rangle\rangle$.

2.4 Escapement Testing

Since interpretation under the abstract escape semantics is guaranteed to terminate, the abstract escape semantics can be used as a basis to infer the escape information for functions in a program at compile-time. We describe escape testing algorithms including a global escape test and a local escape test, in which the abstract functions are used to detect static information about the escape properties of the corresponding functions in a program.

We perform the escape test on each argument of a function call separately. Thus, at any time we are only interested in whether or not a particular single object escapes. Other objects may escape in the result of a function call, but are ignored by our analysis. If a function has n parameters, then escape test will be performed n times, each time treating a different parameter as interesting.

Global Escape Test

We describe a global escape test which provides the global escape information about each function in a program that holds true for every possible application of the function. The global escape test is performed only on a function definition, and thus gives general information about the escape property of that function. To do so, we have to apply the escape semantic value of a function to arguments that cause the greatest escapement possible.

Definition 2.3 (Worst-case Escape Function) For each non-list type τ, we define the abstract function W^τ that corresponds to a function from which every argument escapes.

$$W^\tau \stackrel{\text{def}}{=} \begin{cases} \lambda x_1.\langle x_{1(1)}, \lambda x_2.\langle x_{1(1)} \sqcup x_{2(1)}, \ldots, \lambda x_m.\langle \bigsqcup_{p=1}^{m} x_{i(1)}, err\rangle \ldots\rangle\rangle & m \geq 1 \\ err & m = 0 \end{cases}$$

where m is the number of arguments that a function of type τ can take before returning a primitive value. For each list type of $\tau\ list$, $W^{\tau\ list}$ is defined to be W^τ.

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position i of an interesting parameter, and an abstract escape semantic environment $e\hat{n}v_o$ mapping f to an element of $\hat{D}_o$, the global escape test function G_escape? determines whether the i^{th} parameter of f could possibly escape f or not. It is defined as follows:

$$\mathsf{G_escape?}(f, i, e\hat{n}v_o) = (\hat{O}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_o[x_i \mapsto y_i])_{(1)}$$

where

$$y_i = \langle 1, W^{\tau_i}\rangle,\ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

for all $j \leq n$ and $j \neq i$,

$$y_j = \langle 0, W^{\tau_j}\rangle,\ \text{/* Other parameters are not interesting objects */}$$

and τ_i is the type of the i^{th} parameter of f. Only the i^{th} argument to f is an interesting object and other arguments are not interesting objects. The functional part of each argument causes maximal escapement. Thus, from the result of the global escape test function, we can conclude the following:

- If G_escape?$(f, i, e\hat{n}v_o) = 0$ then we conclude that in any possible application of f to n arguments, the i^{th} argument *does not* escape f.
- If G_escape?$(f, i, e\hat{n}v_o) = 1$ then it means that in some possible application of f to n arguments, the i^{th} argument *could* escape f.

Local Escape Test

Generally, we would like to know if an argument escapes from a particular call to a function. This depends on the values of the arguments of that call. We describe a local escape analysis which provides escape information of a function in a particular context. The local escape test determines the escape behavior of a particular function call, and thus yields more specific results than the global escape test. Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in an application $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and an abstract escape semantic environment $e\hat{n}v_o$ mapping f and the free identifiers within e_1 through e_n to elements of $\hat{D}_o$, the local escape test function L_escape? determines whether the i^{th} parameter of f could escape f during the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$\text{L_escape?}(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = (\hat{O}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_o[x_i \mapsto y_i])_{(1)}$$

where

$$y_i = \langle 1, (\hat{O}_e[\![e_i]\!]\ e\hat{n}v_o)_{(2)}\rangle,\ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

and for all $j \leq n$ and $j \neq i$,

$$y_j = \langle 0, (\hat{O}_e[\![e_j]\!]\ e\hat{n}v_o)_{(2)}\rangle.\ \text{/* Other parameters are not interesting objects */}$$

Only the i^{th} parameter is an interesting object and other parameters are not interesting objects. The functional part of each argument is the functional behavior of each expression e_i. Then, from the result of the local escape test function, we can conclude the following:

- If L_escape?$(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = 0$ then we conclude that the i^{th} argument *does not* escape f in the particular application of f to e_1 through e_n.
- If L_escape?$(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = 1$ then the i^{th} argument *could* escape f in the particular application of f to e_1 through e_n.

Examples

As an example, consider a program defined as follows:

```
letrec  g a b = if (a < b) then 0 else a;

        h c d = if (c < d) then d else 0;

        map f l = if (null l) then nil
                  else cons (f (car l)) (map f (cdr l));

in  ... (map (g 3) [1,3,5]) ... (map (h 3) [1,3,5]) ...
```

We assume that the type of each function is given by

$\texttt{g} : int \to int \to int$
$\texttt{h} : int \to int \to int$
$\texttt{map} : (int \to int) \to int\ list \to int\ list.$

Then, the abstractions g, h, and map of `g`, `h`, and `map` under the abstract escape semantics are defined as follows:

$$\begin{aligned}
g &= \langle 0, \lambda a.\langle a_{(1)}, \lambda b.(\langle 0, err\rangle \sqcup a)\rangle\rangle \\
&= \langle 0, \lambda a.\langle a_{(1)}, \lambda b.a\rangle\rangle \\
h &= \langle 0, \lambda c.\langle c_{(1)}, \lambda d.(\langle 0, err\rangle \sqcup d)\rangle\rangle \\
&= \langle 0, \lambda c.\langle c_{(1)}, \lambda d.d\rangle\rangle \\
map &= \langle 0, \lambda f.\langle f_{(1)}, \lambda l. \\
&\quad \langle 0, err\rangle \sqcup (f\ l) \sqcup ((map_{(2)}\ f)_{(2)}\ l)
\end{aligned}$$

Since map is defined recursively, the meaning of map is found by fixpoint iteration as follows:

$$\begin{aligned}
map^{(0)}\ f\ l &= \langle 0, err\rangle \\
map^{(1)}\ f\ l &= \langle 0, err\rangle \sqcup (f_{(2)}\ l) \sqcup ((map^{(0)}_{(2)}\ f)_{(2)}\ l) \\
&= \langle 0, err\rangle \sqcup (f_{(2)}\ l) \sqcup \langle 0, err\rangle \\
&= f_{(2)}\ l \\
map^{(2)}\ f\ l &= \langle 0, err\rangle \sqcup (f_{(2)}\ l) \sqcup ((map^{(1)}_{(2)}\ f)_{(2)}\ l) \\
&= \langle 0, err\rangle \sqcup (f_{(2)}\ l) \sqcup (f_{(2)}\ l) \\
&= f_{(2)}\ l
\end{aligned}$$

Since $map^{(1)} = map^{(2)}$, we have that $map = \langle 0, \lambda f.\langle f_{(1)}, \lambda l.f_{(2)}\ l\rangle\rangle$. Let $e\hat{n}v_o$ be an abstract escape environment defined as $e\hat{n}v_o = [\texttt{g} \mapsto g, \texttt{h} \mapsto h, \texttt{map} \mapsto map]$.

Global Escape Information

To find the global (i.e. worst-case) escape property of `map`, we apply the global escape test function G_escape?.

$$\mathsf{G_escape?}(\texttt{map}, 1, \hat{env}_o) = (\hat{O}_e[\![\texttt{map f l}]\!]\hat{env}'_o)_{(1)} = 0$$

where

$$\hat{env}'_o = \hat{env}_o[\texttt{f} \mapsto \langle 1, W^{int\to int}\rangle, \texttt{l} \mapsto \langle 0, W^{int\,list}\rangle],$$
$$W^{int\to int} = \lambda x.\langle x_{(1)}, err\rangle,\ W^{int\,list} = err.$$

Thus, we can conclude that the first parameter `f` of the function `map` can never escape `map` globally. And,

$$\mathsf{G_escape?}(\texttt{map}, 2, \hat{env}_o) = (\hat{O}_e[\![\texttt{map f l}]\!]\hat{env}'_o)_{(1)} = 1$$

where

$$\hat{env}'_o = \hat{env}_o[\texttt{f} \mapsto \langle 0, W^{int\to int}\rangle, \texttt{l} \mapsto \langle 1, W^{int\,list}\rangle],$$
$$W^{int\to int} = \lambda x.\langle x_{(1)}, err\rangle,\ W^{int\,list} = err.$$

This means that the second parameter `l` of the function `map` could escape in some situation. Thus, we cannot say that `l` never escape `map` globally.

Local Escape Information

To determine the local escape property of `map` in a particular context, we apply the local escape test function L_escape?.

$$\mathsf{L_escape?}(\texttt{map}, 2, (\texttt{g 3}), [1,3,5], \hat{env}_o) = (\hat{O}_e[\![\texttt{map f l}]\!]\hat{env}'_o)_{(1)} = 0$$

where

$$\hat{env}'_o = \hat{env}_o[\texttt{f} \mapsto \langle 0, (g_{(2)}\langle 0, err\rangle)_{(2)}\rangle, \texttt{l} \mapsto \langle 1, err\rangle].$$

Thus, we can conclude that the second parameter `l` of the function `map` does not escape locally in (`map (g 3) [1,3,5]`).

$$\mathsf{L_escape?}(\texttt{map}, 2, (\texttt{h 3}), [1,3,5], \hat{env}_o) = (\hat{O}_e[\![\texttt{map f l}]\!]\hat{env}'_o)_{(1)} = 1$$

where

$$\hat{env}'_o = \hat{env}_o[\texttt{f} \mapsto \langle 0, (h_{(2)}\langle 0, err\rangle)_{(2)}\rangle, \texttt{l} \mapsto \langle 1, err\rangle].$$

So, we can conclude that the second parameter `l` of the function `map` does escape locally in (`map (h 3) [1,3,5]`).

2.5 Improving Precision of Escapement

In the abstract escape semantics described in the previous section, a list is treated as a single object. Once an interesting object is put in a list, we have assume that the object remains in the list. Thus, no matter how many times `cdr` is applied to the list, we assume that the interesting object remains. Furthermore, any time `car` is applied to the list, we assume that the interesting object might be contained in the result.

Consider, for example, the following functions:

```
letrec  f x y = car(cons x (cons y nil));
        g x y = cdr(cons x (cons y nil));
in ... cons (f 1 2) (g 1 2) ...
```

where `f` is of type $int \rightarrow int \rightarrow int$ and `g` is of type $int \rightarrow int \rightarrow int\ list$. Under the abstract escape semantics, the definitions of values f and g of `f` and `g` are the same.

$$\begin{aligned} f &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup (y \sqcup \langle 0, err\rangle)\rangle\rangle \\ &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\ &= g \end{aligned}$$

From the escape analysis based on the abstract escape semantics, we conclude only that both `x` and `y` could escape `f` and `g`. In fact, for the function `f`, only the first parameter `x` escapes `f`, but the second parameter `y` never escapes `f`. Similarly, only the second parameter `y` actually escapes `g`, but the first parameter `x` never escapes `g`.

In this section, we present a method for improving the precision of escape information that is obtainable through the escape analysis using the position information of interesting objects in a list structure.

2.5.1 Positions of a List

The information about where an interesting object occurs in a list will provide a more accurate approximation of escapement. We describe how to include information about object's position within a list in the escape semantics, in order to keep track (approximately) of what position(first element, second element, etc.) in the top spine of the list an object might occur first. The notion of position in a list is shown in Figure 2.6.

Definition 2.4 (Positions of a List) An object is said to be at the *position* of i in a list L if some part of or all of the object *only* resides in the sublist of L whose root cell is specified by $\texttt{cdr}^i\ L$ for some $i \geq 0$, i.e. In other word, *no* part of the object is contained in the first $(i-1)$ positions of L.

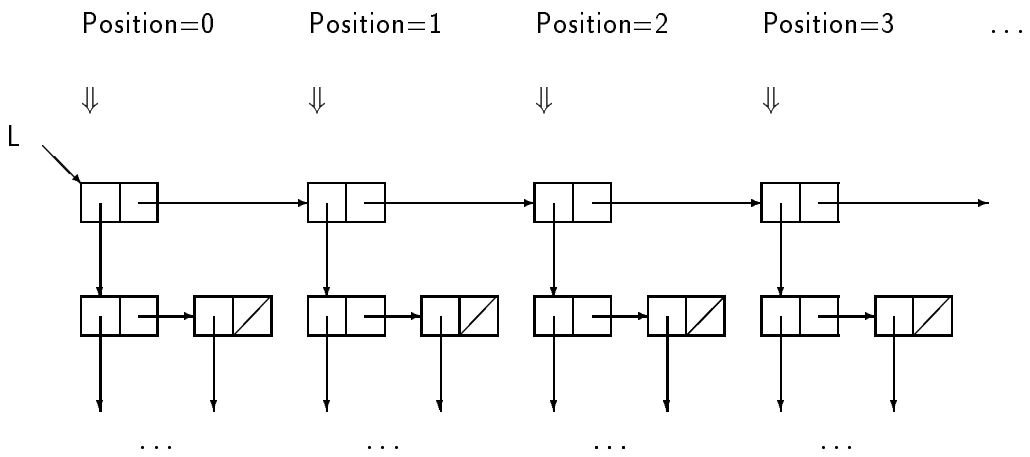

Figure 2.6: Positions of a List

Since no finite bound on the length of lists can be computed at compile time, we impose a bound on the positions in the list that we are willing to keep track of. Beyond this bound, we assume that the object remains in the list.

2.5.2 Improved Abstract Escape Semantics

We present another safe and computable abstraction of the exact escape semantics that gives escape information with more precision than the abstract escape semantics previously described.

Improved Abstract Escape Domains

The value of each expression in the improved escape semantic domain indicates either that no part of an interesting object is contained in the value of the expression ("non-escape"), or that some part of an interesting object may be contained at some position in the value of the expression ("possible escape and where"). The improved abstraction of the exact escape semantics is done by extending the basic abstract escape domain to including position information. The basic improved abstract escape domain, $\tilde{B}_o$, for some fixed p is a $(p+2)$-element domain of pairs as shown in Figure 2.7. The ordering on pairs in $\tilde{B}_o$ is defined as follows:

$$\langle 0,0\rangle \sqsubseteq \langle 1,p\rangle \sqsubseteq \langle 1,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,1\rangle \sqsubseteq \langle 1,0\rangle$$

The interpretation of elements of $\tilde{B}_o$ is defined as follows:

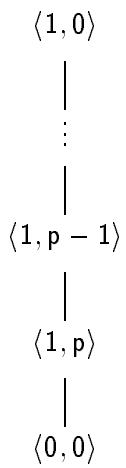

Figure 2.7: The Improved Abstract Basic Escape Domain

- $\langle 1, k \rangle$: Some part of or all of an interesting object *may be* contained in the value of the expression only at a position greater than or equal to $\geq k$. (not at a position less than k)
- $\langle 0, 0 \rangle$: *No* part of any interesting object *is* contained in the value of the expression.

The improved abstract escape domain $\tilde{D}_o$ is an improved abstraction of D_o, and the improved abstract escape environment $\tilde{E}_o$ is a domain of functions mapping identifiers to their abstract escape meanings. They are defined as follows:

$$\begin{aligned} \tilde{D}_o &= \sum_{\tau} \tilde{D}_o^{\tau} \qquad \text{/* Improved abstract escape semantic domain */} \\ \tilde{E}_o &= Id \rightarrow \tilde{D}_o \quad \text{/* Domain of improved abstract escape environments */} \end{aligned}$$

The improved abstract escape subdomain $\hat{D}_o^{\tau}$ for expression of type τ is defined as follows:

$$\begin{aligned} \tilde{D}_o^{int} &= \tilde{B}_o \times \{err\} && \text{improved abstract subdomain for integers} \\ \tilde{D}_o^{bool} &= \tilde{B}_o \times \{err\} && \text{improved abstract subdomain for booleans} \\ \tilde{D}_o^{\tau_1 \rightarrow \tau_2} &= \tilde{B}_o \times (\tilde{D}_o^{\tau_1} \rightarrow \tilde{D}_o^{\tau_2}) && \text{improved abstract subdomain for functions} \\ \tilde{D}_o^{\tau\ list} &= \tilde{D}_o^{\tau} && \text{improved abstract subdomain for lists} \end{aligned}$$

Improved Abstract Escape Semantic Functions

We now introduce improved abstract escape semantic functions as follows:

$$
\begin{array}{lcl}
\tilde{O}_c[\![c]\!] & = & \langle\langle 0,0\rangle, err\rangle, \;\; c \in \{\ldots,-1,0,1,\ldots,\texttt{true},\texttt{false}\} \\
\tilde{O}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (The bottom element in } \tilde{D}_o^\tau \text{)} \\
\tilde{O}_c[\![\texttt{cons}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\text{push}(x,y)\rangle\rangle \\
 & & \text{where} \\
 & & \text{push}(x,y) = \text{if } (x_{(1)(1)} = 1) \text{ then } \langle\langle 1,0\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
 & & \text{elseif } (y_{(1)(1)} = 1) \text{ then } \langle\langle 1, \min(y_{(1)(2)}+1, p)\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
 & & \text{else } \langle\langle 0,0\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
\tilde{O}_c[\![\texttt{car}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\ \text{pick}(x)\rangle \\
 & & \text{where pick}(z) = \text{if } (z_{(1)(2)} > 0) \text{ then } \langle\langle 0,0\rangle, z_{(2)}\rangle \text{ else } z \\
\tilde{O}_c[\![\texttt{cdr}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\ \text{rest}(x)\rangle \\
 & & \text{where } \ \text{rest}(z) = \langle\langle z_{(1)}, \max(z_{(2)}-1, 0)\rangle, z_{(2)}\rangle \\
\tilde{O}_c[\![\texttt{null}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle\langle 0,0\rangle, err\rangle
\end{array}
$$

Figure 2.8: Improved Abstract Escape Semantic Functions

$$
\begin{array}{lcll}
\tilde{O}_c & : & Con \to \tilde{D}_o & \text{/* Improved semantic function for constants */} \\
\tilde{O}_e & : & Exp \to \tilde{E}_o \to \tilde{D}_o & \text{/* Improved semantic function for expressions */} \\
\tilde{O}_{pr} & : & Program \to \tilde{D}_o & \text{/* Improved semantic function for programs */}
\end{array}
$$

The improved abstract escape semantic functions are given in Figure 2.8 and Figure 2.9.

The improved abstract value of `cons` takes two arguments, and returns an improved escape pair of their least upper bound by updating the position information of an interesting object in the result list. The improved abstract value of `car` returns its argument according to the position of an interesting object. The improved abstract value of `cdr` also updates the position information appropriately. Note that the improved abstract escape semantic function $\tilde{O}_c$ for `cons`, `car` and `cdr` provides more precise escape information than the abstract escape semantic function $\hat{O}_c$. $e\hat{n}v_o$ is an improved abstract escape environment in $\tilde{E}_o$, and $nul\tilde{l}env_o$ is an improved abstract escape environment that maps every identifier on to the least element of its improved abstract escape semantic domain.

Safety and Termination

To show that the improved abstract escape semantics will never produce wrong escape information with respect to the exact escape semantics, we introduce a notion of *safety* with respect to exact escape information that relates the improved abstract escape semantics with

$$
\begin{aligned}
\tilde{O}_e[\![c]\!]e\tilde{n}v_o &= \tilde{O}_c[\![c]\!] \\
\tilde{O}_e[\![x]\!]e\tilde{n}v_o &= e\tilde{n}v_o[\![x]\!] \\
\tilde{O}_e[\![e_1 + e_2]\!]e\tilde{n}v_o &= \langle\langle 0,0\rangle, err\rangle \ /^* \text{ same for } e_1 - e_2 \text{ and } e_1 = e_2 \ ^*/ \\
\tilde{O}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\tilde{n}v_o &= (\tilde{O}_e[\![e_2]\!]e\tilde{n}v_o) \sqcup (\tilde{O}_e[\![e_3]\!]e\tilde{n}v_o) \\
\tilde{O}_e[\![e_1 e_2]\!]e\tilde{n}v_o &= (\tilde{O}_e[\![e_1]\!]e\tilde{n}v_o)_{(2)}\ (\tilde{O}_e[\![e_2]\!]e\tilde{n}v_o) \\
\tilde{O}_e[\![\texttt{lambda}(x).e]\!]e\tilde{n}v_o &= \langle \tilde{V}, \lambda y.\tilde{O}_e[\![e]\!]e\tilde{n}v_o[x \mapsto y]\rangle
\end{aligned}
$$

where

$\tilde{V} = \langle 0,0\rangle \sqcup (\bigsqcup_{z \in F} (env_o[\![z]\!])_{(1)})$ and

F = Set of all free identifiers in $(\texttt{lambda}(x).e)$.

$$
\tilde{O}_e[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n; \texttt{in } e]\!]e\tilde{n}v_o = \tilde{O}_e[\![e]\!]e\tilde{n}v'_o
$$

where $e\tilde{n}v'_o = e\tilde{n}v_o[x_1 \mapsto \tilde{O}_e[\![e_1]\!]e\tilde{n}v'_o, \ldots, x_n \mapsto \tilde{O}_e[\![e_n]\!]e\tilde{n}v'_o]$

$$
\tilde{O}_{pr}[\![pr]\!] = \tilde{O}_e[\![pr]\!]nul\tilde{l}env_o
$$

Figure 2.9: Improved Abstract Escape Semantic Functions

the exact escape semantics. Let u and v be values of an expression e of type τ or τ *list* in the improved abstract escape domain $\tilde{D}_o$ and the exact escape domain D_o, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that the improved abstract escape semantic value u is a *safe* approximation (with respect to exact escape information) for the exact escape semantic value v iff

$$
(\bigsqcup_{p \textbf{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k)} p_{(1)}) \sqsubseteq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(1)}
$$

and

$$
(MIN_{p \textbf{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k) \& p_{(1)}=1} \text{Position of } p) \geq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(2)}
$$

for all $k \leq n$ where s_i is a *safe* approximation for t_i for all $i \leq k$.

Theorem 2.3 (Safety) *For any expression e, and environments env_o and $e\tilde{n}v_o$ such that for all y, $e\tilde{n}v_o[\![y]\!]$ is safe for $env_o[\![y]\!]$, $\tilde{O}_e[\![e]\!]e\tilde{n}v_o$ is safe (with respect to escape information) for $O_e[\![e]\!]env_o$.*

Proof : We can prove by structural induction on expression e.

I. Base Case:

1. $e = c$: $\tilde{O}_e[\![c]\!]e\tilde{n}v_o = \tilde{O}_c[\![c]\!]$ and $O_e[\![c]\!]env_o = O_c[\![c]\!]$. For $c \in \{\ldots,$`-1,0,1,...true, false, nil, null` $\}$, $O_c[\![c]\!] = \tilde{O}_c[\![c]\!]$ and thus it clearly holds. For $c =$ `cons`, it holds because, if x_1 and x_2 are safe for y_1 and y_2, respectively, push(x, y) is also safe for a list consisting of y_1 and y_2. It holds for $c =$ `car`, `cdr`, because if x is safe for y then pick(x) is safe for first(y) and rest(x) is safe for second(y).

2. $e = x$: $\tilde{O}_e[\![x]\!]e\tilde{n}v_o = e\tilde{n}v_o[\![x]\!]$ and $O_e[\![x]\!]env_o = env_o[\![x]\!]$. Since for all y $e\tilde{n}v_o[\![y]\!]$ is safe for $env_o[\![y]\!]$, it clearly holds.

(II) Structural Induction Step: Assume that $\tilde{O}_e[\![e]\!]e\tilde{n}v_o$ is safe for $O_e[\![e]\!]env_o$ for expressions such as e_0, e_1, e_2, e_3 and e_n. (structural induction hypothesis) Then, we show that $\tilde{O}_e[\![e]\!]e\tilde{n}v_o$ is safe for $O_e[\![e]\!]env_o$ for $e = e_1 + e_2$, `if` e_1 `then` e_2 `else` e_3, e_1e_2, `lambda`$(x).e$, and `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0. This can be proved in an exactly similar way to the proof of safety of the abstract escape semantics. □

The effectiveness of interpretation under the improved abstract escape semantics means that it terminates on every valid program at compile-time.

Theorem 2.4 (Termination) *For any (finite) program $pr \in Program$, $\tilde{O}_{pr}[\![pr]\!]$ are computable, i.e. always terminates in finite number of steps.*

Proof : Every functional in the improved escape domain that is defined through the abstract improved escape semantic functions is composed of the operators such as the least upper bound operator $\sqcup$, push, pick, and rest. *pick* and *rest* are monotonic operators. Since the composition of monotonic functions is also monotonic, every functional is monotonic. Furthermore, each subdomain $\tilde{D}^\tau$ is finite. □

Precision Improvement

The precision improvement of the improved abstract escape semantics over the abstract escape semantics means that the escape information obtained by the improved abstract escape semantics always implies that obtained by the abstract escape semantics, but not always vice-versa.

Theorem 2.5 (Precision Improvement) *The abstract escape semantics is equivalent in its information content to the improved abstract escape semantics with $p = 0$. Thus, the improved abstract escape semantics with some $p > 0$ provides more precise escape information than the abstract escape semantics.*

Proof : We introduce the notion of *equivalence* with respect to escapement information content which relates the abstract escape semantics to the improved abstract escape seman-

tics with $p = 0$. Let u and v be values of an expression e of type τ in $\hat{D}_o$ and $\tilde{D}_o$ with $p = 0$, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that u is *equivalent* (with respect to escape information) to v iff

$$(\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)} = (\mathrm{NAP}_k(v, t_1, \ldots, t_k))_{(1)(1)}$$

for all $k \leq n$ where s_i is *equivalent* to t_i for all $i \leq k$. Then, for all expression e and environments $e\hat{n}v_o$ and $e\tilde{n}v_o$ with $p = 0$ such that for all y, $e\hat{n}v_o[\![y]\!]$ is semantically equivalent (with respect to escape information) to $e\tilde{n}v_o[\![y]\!]$ with $p = 0$, we have that $\hat{O}_e[\![e]\!]e\hat{n}v_o$ is semantically equivalent to $\tilde{O}_e[\![e]\!]e\tilde{n}v_o$ with $p = 0$. This can be proved by structural and fixpoint inductions.

We introduce the notion of *improvement* with respect to escapement information content which relates the improved abstract escape semantics with $p > 0$ to the improved abstract escape semantics with $p = 0$. Let u and v be values of an expression e of type τ in $\tilde{D}_o$ with $p > 0$ and $\tilde{D}_o$ with $p = 0$, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that u is *improved* (with respect to escape information) over v iff

$$(\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(1)} \sqsubseteq (\mathrm{NAP}_k(v, t_1, \ldots, t_k))_{(1)(1)}$$

and

$$(\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(2)} \sqsubseteq (\mathrm{NAP}_k(v, t_1, \ldots, t_k))_{(1)(2)} = 0$$

for all $k \leq n$ where s_i is *improved* over t_i for all $i \leq k$. Then, for all expression e and environments $e\tilde{n}v_o$ with $p > 0$ and $e\tilde{n}v_o$ with $p = 0$ such that for all y, $e\tilde{n}v_o[\![y]\!]$ with $p > 0$ is improved over $e\tilde{n}v_o[\![y]\!]$ with $p = 0$, we have that $e\tilde{n}v_o[\![y]\!]$ with $p > 0$ is improved over $\tilde{O}_e[\![e]\!]e\tilde{n}v_o$ with $p = 0$. This can be proved by structural and fixpoint inductions. □

The relationship among the standard semantics (SS), the non-standard exact escape semantics (ES), the abstract escape semantics (AES) and the improved abstract escape semantics (IAES) is shown in Figure 2.10.

2.5.3 Improved Escapement Testing

We describe global and local escape testing algorithms based on the improved abstract escape semantics.

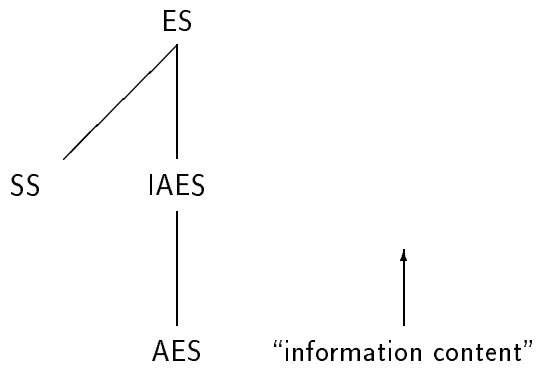

Figure 2.10: Relationship among Standard and Escape semantics

Improved Global Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position i of an interesting parameter, and an improved abstract escape semantic environment $e\tilde{n}v_o$ mapping f to an element of $\tilde{D}_o$, the improved global escape test function G_iescape? which determines whether the i^{th} parameter of f could possibly escape f in any application of f or not is defined as follows:

$$\mathsf{G_iescape?}(f, i, e\tilde{n}v_o) = (\tilde{O}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\tilde{n}v_o[x_i \mapsto y_i])_{(1)(1)}$$

where

$$y_i = \langle\langle 1, 0\rangle, W^{\tau_i}\rangle,\ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

and for all $j \leq n$ and $j \neq i$,

$$y_j = \langle\langle 0, 0\rangle, W^{\tau_j}\rangle,\ \text{/* Other parameters are not interesting objects */}$$

and τ_i is the type of the i^{th} parameter of f. Only the i^{th} argument to f is an interesting object and other arguments are not interesting objects. Since the whole i^{th} argument is interesting, its position value is set 0. The result of the global improved escape test function is interpreted as follows:

- If $\mathsf{G_iescape?}(f, i, e\tilde{n}v_o) = 0$ then we conclude that the i^{th} argument *does not* escape f in any possible application of f to n arguments.
- If $\mathsf{G_iescape?}(f, i, e\tilde{n}v_o) = 1$ then we *cannot* say that the i^{th} argument does not escape f in any possible application of f to n arguments.

Improved Local Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in an application context $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and an improved abstract escape semantic environment $e\tilde{n}v_o$ mapping f and the free identifiers within e_1 through e_n to elements of $\tilde{D}_o$, the improved local escape test function L_iescape? which determines whether the i^{th} parameter of f could escape f locally in the evaluation of $f\ e_1\ \ldots\ e_n$ is defined as follows:

$$\mathsf{L_iescape?}(f, i, e_1, \ldots, e_n, e\tilde{n}v_o) = (\tilde{O}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\tilde{n}v_o[x_i \mapsto y_i])_{(1)(1)}$$

where

$$y_i = \langle\langle 1, 0\rangle, (\tilde{O}_e[\![e_i]\!]\ e\tilde{n}v_o)_{(2)}\rangle\ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

and for all $j \leq n$ and $j \neq i$,

$$y_j = \langle\langle 0, 0\rangle, (\tilde{O}_e[\![e_j]\!]\ e\tilde{n}v_o)_{(2)}\rangle.\ \text{/* Other parameters are not interesting objects */}$$

The result of the local improved escape test function is interpreted as follows:

- If $\mathsf{L_iescape?}(f, i, e_1, \ldots, e_n, e\tilde{n}v_o) = 0$ then we conclude that the i^{th} argument *does not* escape f in the particular application of f to e_1 through e_n.
- If $\mathsf{L_iescape?}(f, i, e_1, \ldots, e_n, e\tilde{n}v_o) = 1$ then we cannot say that the i^{th} argument *does not* escape f in the particular application of f to e_1 through e_n.

Examples

As an example, consider the functions given before:

```
letrec
   f x y = car(cons x (cons y nil));
   g x y = cdr(cons x (cons y nil));
in ...
```

where f is of type $int \rightarrow int \rightarrow int$ and g is of type $int \rightarrow int \rightarrow int\ list$. Under the improved abstract escape semantics, the definitions of the values f and g of f and g are expressed as follows:

$$\begin{aligned} f &= \langle\langle 0, 0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\text{pick}(\text{push}(x, \text{push}(y, \langle\langle 0, 0\rangle, err\rangle)))\rangle\rangle \\ g &= \langle\langle 0, 0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\text{rest}(\text{push}(x, \text{push}(y, \langle\langle 0, 0\rangle, err\rangle)))\rangle\rangle \end{aligned}$$

Let $e\tilde{n}v_o$ be defined as $e\tilde{n}v_o = [\mathtt{f} \mapsto f, \mathtt{g} \mapsto g]$

$$\mathsf{G_escape?}(\mathtt{f}, 2, e\tilde{n}v_o) = (\tilde{O}_e[\![\mathtt{f\ \ x\ \ y}]\!]e\tilde{n}v'_o)_{(1)(1)} = 0$$

where $e\tilde{n}v'_o = e\tilde{n}v_o[\mathtt{x} \mapsto \langle\langle 0,0\rangle, err\rangle, \mathtt{y} \mapsto \langle\langle 1,0\rangle, err\rangle]$. Thus, we can conclude that the second parameter `y` of `f` does not escape `f` globally. Similarly,

$$\mathsf{G_escape?}(\mathtt{g}, 1, e\tilde{n}v_o) = (\tilde{O}_e[\![\mathtt{g\ \ x\ \ y}]\!]e\tilde{n}v'_o)_{(1)(1)} = 0$$

where $e\tilde{n}v'_o = e\tilde{n}v_o[\mathtt{x} \mapsto \langle\langle 1,0\rangle, err\rangle, \mathtt{y} \mapsto \langle\langle 0,0\rangle, err\rangle]$. Thus, we can conclude that the first parameter `x` of `g` does not escape `g` globally.

2.6 Complexity of Escape Analysis

The abstract interpretation framework for the escape analysis that deals with higher-order functional languages with lists is very similar to the framework for strictness analysis for higher-order functional languages without non-flat domains (lists). Both analyses use a two-element domain as their basic abstract domains, respectively. Thus, the order of time complexity of escape analysis is the same as that of strictness analysis for higher-order languages with non-flat domains, which is *exponential* in the number of arguments to the function being analyzed in case of non-higher-order functions. In case of a higher-order function, the complexity is much worse than exponential in the number of arguments to the function being analyzed, and depends on the types of its arguments. However, the improved escape analysis uses a $(p+2)$-element domain as its basic abstract domain. So, the order of time complexity of improved escape analysis is higher than, but comparable to, that of strictness analysis for higher-order languages with non-flat domains. The abstract interpretation framework for strictness analysis for higher-order functional languages with flat domains uses a k-element domain as its abstract domain where k is fixed but greater than 2. Thus, when compared with the complexity of strictness analysis for higher-order functional languages with non-flat domains, the complexity of escape analysis is less, and the complexity of improved escape analysis is similar to that.

Fortunately, as is true for many analyses, worst-case situations rarely occur in practice and the number of iterations required is typically small. Furthermore, the average number of arguments to functions does not grow with the size of a program, and thus if an arbitrary bound is placed on the number of arguments, then the analysis based on the abstract escape semantics can be shown to be linear in program size.

Chapter 3

Refinements of Escape Analysis

The escape analysis described in the previous chapter is concerned with the escapement properties of arguments and local objects of a function. These objects are treated *as a whole* with respect to the activation of the function call. That is, even when only some part of such an object escapes, the escape analysis indicates only that the object escapes. Thus, for structured objects such as lists and trees, the escape information that is obtainable through the escape analysis is rather coarse because it does not specify the escaping substructure even when only some part of a structured object escapes. Such refined escape information, if inferred at compile-time, can be useful for more efficient management of storage for structured objects.

In this chapter, we present a method for computing more refined escape information for a higher-order, monomorphic, strict functional language. This method is based on a compile-time semantic analysis called *refined escape analysis* which is an extension of the escape analysis and indicates at compile-time how much of an object such as argument or local object of a function outlives the activation of the function call. First, we introduce a non-standard denotational semantics called *refined escape semantics* that describes the actual refined escape behavior, but is incomputable at compile-time. A safe and computable abstraction of the exact refined escape semantics is then presented. Based on the *abstract refined escape semantics*, we describe the escape testing algorithms which determine refined escape information. Another safe and computable abstraction of the exact escape semantics, called *improved abstract refined escape semantics*, that improves the precision of refined escape information is also presented using the position information of objects in a list structure. Finally, the relationship between escape analysis and refined escape analysis with respect to their information content is discussed.

3.1 Refined Escapement of Objects

The escape model described in the previous chapter is concerned with the escapement of arguments each of which is treated as a whole object. Thus, for structured data objects such as lists, the escape information that can be obtainable through the escape analysis is rather weak. Consider, for example, the append function defined as follows:

```
letrec  append x y = if (null x) then y
                     else cons (car x) (append (cdr x) y);
in ... append [1,2,3] [4,5,6] ...
```

We assume that `append` is typed as $int\ list \rightarrow int\ list \rightarrow int\ list$. From the escape analysis described in the previous chapter, we can only conclude that both the first and the second parameters of `append` could escape as follows: The definition of the escape semantic value *append* of `append` (shown uncurried for convenience) is:

$$append\ x\ y\ =\ y \sqcup (x \sqcup append\ x\ y)$$

The meaning of *append* is found by fixpoint iteration as follows:.

$$append = \langle 0, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle.$$

Let $e\hat{n}v_o$ be defined as $e\hat{n}v_o = [\texttt{append} \mapsto append]$ Then,

$$\begin{aligned} \texttt{G_escape?}(\texttt{append}, 1, e\hat{n}v_o) &= (\hat{O}_e[\![\texttt{append x y}]\!]e\hat{n}v'_o)_{(1)} \\ &= (\langle\langle 0, err\rangle \sqcup (\langle\langle 1, err\rangle\rangle)_{(1)} = 1 \end{aligned}$$

where $e\hat{n}v'_o = e\hat{n}v_o[\texttt{x} \mapsto \langle 1, err\rangle, \texttt{y} \mapsto \langle 0, err\rangle]$. And,

$$\begin{aligned} \texttt{G_escape?}(\texttt{append}, 2, e\hat{n}v_o) &= (\hat{O}_e[\![\texttt{append x y}]\!]e\hat{n}v'_o)_{(1)} \\ &= (\langle 0, err\rangle \sqcup (\langle 1, err\rangle))_{(1)} = 1 \end{aligned}$$

where $e\hat{n}v'_o = e\hat{n}v_o[\texttt{x} \mapsto \langle 0, err\rangle, \texttt{y} \mapsto \langle 1, err\rangle]$. Thus, we conclude that `append` returns its first and second arguments `x` and `y`. In fact, however, only some portion of the second parameter `y` of the function `append` actually escapes, but some portion does not escape.

In this chapter we will describe a method for inferring more precise escape information for structured objects such as lists. We first define the notion of partial escapement which specifies more a refined notion of escapement of objects as shown in Figure 3.1.

Definition 3.1 (Global/Local Refined Escapement) Given a function f with n formal parameters and m locally defined objects, the i^{th} formal parameter or local object is said to

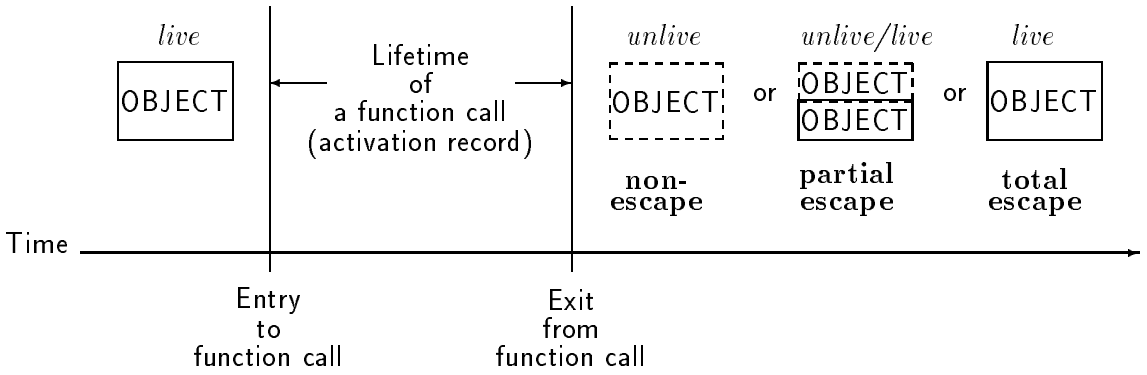

Figure 3.1: Refined Escapement of Objects

- *partially-escape* the function call to f *globally* if, in *some* possible application of f to n arguments, *some* part of the corresponding actual argument or local object outlives the activation of the function call, but the rest of the corresponding actual argument or local object does not outlive the activation of the function call.
- *partially-escape* the function call to f *locally* in $(f\ e_1\ \ldots e_n)$ if, in the particular function application of $(f\ e_1\ \ldots e_n)$, *some* part of the corresponding actual argument or local object outlives the activation of the function call, but the rest of the corresponding actual argument or local object does not outlive the activation of the function call.

When *all* of an object outlives a function call globally (or locally), we say that the object *totally-escapes* the function call globally (or locally).

We will develop a method for higher-order, monomorphic, strict functional languages to answer the following questions at compile-time:

- Given a function, which parameter or local object defined inside the function escapes the function call globally and to what extent (i.e. how much of the object escapes)?
- Given a function in a particular application context, which parameter or local object defined inside the function escapes that particular function call and to what extent?

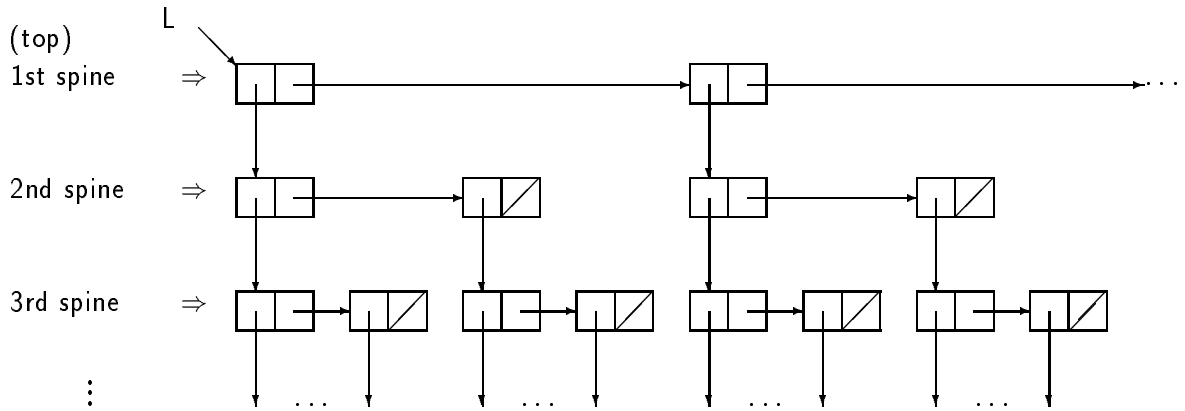

Figure 3.2: Spines of a List

3.2 Refined Escape Analysis

3.2.1 Spines of a List

In a refined escape model, we divide objects into their subcomponents and we are concerned with how much of an object, such as a parameter or local object, escapes the function. We describe a way of partitioning list objects into subcomponents based on the notion of spines of a list as is shown in Figure 3.2.

Definition 3.2 (Spines of a List) Given a list L and some $i \geq 1$, the *top* i^{th} spine of L is defined as the set of cons cells accessible by a sequence of operations consisting of `car` and `cdr` where the number of occurrences of `car` is $(i-1)$. Similarly, given a list L with d spines and some $j \geq 1$, the *bottom* j^{th} spine of L is defined as the *top* $(d-j+1)$ spine of L.

We have chosen to analyze the escape properties of lists in terms of their *spines* for two reasons:

- It is an approximation to the run-time behavior that allows a compile-time analysis.
- It reflects the programming style commonly used for strongly typed languages, such as ML, in which lists are homogeneous (all elements have the same type) and functions (such as append, map, etc.) often operate over complete spines of lists.

The first point reflects our inability to determine precisely, without actually running the program, which individual cells of a list might escape. To form a terminating compile-time

escape analysis, one must choose an approximation of program behavior. The second point reflects our belief that the spines are a good choice of approximation, since the cells of each spine of a list tend to be treated identically. Many functions, such as append, reduce, map, length, etc., operate on all cells of a spine. Many other functions have the form:

```
f l = if predicate(car l) then ... else f (cdr l)
```

or

```
g l x = if x=n then ... else g (cdr l) (arith-op x)
```

In general, it is impossible to determine at compile time when the recursion will bottom out. One simply has to assume that all cells in the spine of the list L will be visited.

3.2.2 Exact Refined Escape Semantics

We introduce an exact, but incomputable, non-standard denotational semantics called *refined escape semantics*, which exactly describes the operational notion of refined escapement for functions in a program. Our refined escape semantics is also defined in terms of a single interesting object, because we consider that each parameter or local object will be analyzed separately to determine its refined escape behavior.

Representing Refined Escape Information

The meaning we will attach to the syntax is the refined information about escaping objects. For each expression, we want its corresponding value in the extended escape semantic domain to be able to tell us how much of an interesting object is contained in the value of the expression ("partial-escape"). Under the non-standard refined escape semantics, we represent the meaning of an expression as a pair, called a *refined escape pair*,

1. whose first element denotes the containment and the extent of an interesting object in the value of the expression, and

2. whose second element denotes the functional behavior of the expression defined over the escape pairs when the expression itself is applied to another expression.

For a non-list type expression, the corresponding value in the non-standard refined escape semantic domain D_p has two components: The first component is an element of a domain

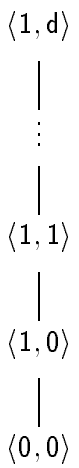

Figure 3.3: The Basic Refined Escape Domain

called a *basic refined escape domain*, B_p, for some fixed d is a $(d+2)$-element domain of pairs as shown in Figure 3.3. The ordering on pairs in B_p is defined as follows:

$$\langle 0,0\rangle \sqsubseteq \langle 1,0\rangle \sqsubseteq \langle 1,1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,d-1\rangle \sqsubseteq \langle 1,d\rangle$$

The interpretation of elements of B_p is defined as follows:

- $\langle 1,j\rangle$: Only the bottom j spines of an interesting object *is* contained in the value of the expression, and the rest of an interesting object is not contained. (If an interesting object is not a list then j will always be 0, which means that an indivisible interesting object *is* contained in the value of the expression.)
- $\langle 0,0\rangle$: *No* part of any interesting object *is* contained in the value of the expression whose evaluation is ever terminating.

The second component is a function over D_p, whose meaning is the functional behavior of the expression when the expression itself is applied to another expression. For expressions which have no higher-order behavior, err, which is a function that can never be applied, is used. For a list type expression, the value in the escape semantic domain is a list that consist of the values of its components in the refined escape semantic domains.

Refined Escape Semantic Domains

The refined escape semantic domain D_p and the domain of refined escape environments E_p are defined as follows:

$$
\begin{array}{lcll}
D_p & = & \sum_{\tau} D_p^{\tau} & /^* \text{ Refined escape semantic domain } ^*/ \\
E_p & = & Id \rightarrow D_p & /^* \text{ Domain of refined escape environments } ^*/
\end{array}
$$

The refined escape domain D_p is a sum domain of each subdomain of each type. The refined escape subdomain D_p^{τ} for expressions of type τ is defined as follows:

$$
\begin{array}{lcll}
D_p^{int} & = & B_p \times \{err\} & \text{subdomain for integers} \\
D_p^{bool} & = & B_p \times \{err\} & \text{subdomain for booleans} \\
D_p^{\tau_1 \rightarrow \tau_2} & = & B_p \times (D_p^{\tau_1} \rightarrow D_p^{\tau_2}) & \text{subdomain for functions of type } \tau_1 \rightarrow \tau_2 \\
D_p^{\tau\ list} & = & (B_p \times \{err\}) + (D_p^{\tau} \times D_p^{\tau\ list}) & \text{subdomain for lists of type } \tau\ list
\end{array}
$$

Refined Escape Semantic Functions

The non-standard refined escape semantic functions are defined as follows:

$$
\begin{array}{lcll}
P_c & : & Con \rightarrow D_p & /^* \text{ Refined escape function for constants } ^*/ \\
P_e & : & Exp \rightarrow E_p \rightarrow D_p & /^* \text{ Refined escape function for expressions } ^*/ \\
P_{pr} & : & Program \rightarrow D_p & /^* \text{ Refined escape function for programs } ^*/
\end{array}
$$

The semantic equations for the refined escape semantic functions P_c that gives non-standard refined escape meaning to constants, P_e that gives non-standard refined escape meaning to expressions, and P_{pr} that gives non-standard refined escape meaning to programs, are expressed in Figure 3.4. Note that **Oracle** is also used to resolve the exact escape behavior of the conditional expression `if`. env_p is any exact refined escape environment in E_p, and $nullenv_p$ is a refined escape environment that maps every identifier to the least element of its refined escape semantic domain.

3.2.3 Abstract Refined Escape Semantics

We present a safe and computable abstraction of the exact refined escape semantics defined in the last section that allows an approximation of the exact escape behavior of functions to be found at compile-time.

We safely approximate the exact refined escape semantics by abstracting the refined escape semantic subdomains for list type expressions, and by approximating the refined escape semantic functions. For each expression, its corresponding value in the abstract extended escape semantic domain tells us how much of an interesting object may be contained in the value of the expression ("maybe partial-escape"). The abstract basic refined escape domain, $\hat{B}_p$ for some fixed d is a $(d+2)$-element domain that is similar to the basic refined

$$
\begin{aligned}
P_c[\![c]\!] &= \langle\langle 0,0\rangle, err\rangle, c = \{\ldots, \texttt{-1}, \texttt{0}, \texttt{1}, \ldots, \texttt{true}, \texttt{false}, \texttt{nil}^{\tau\, list}\} \\
P_c[\![\texttt{cons}]\!] &= \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\ \mathrm{pair}(x,y)\rangle\rangle \\
P_c[\![\texttt{car}]\!] &= \langle\langle 0,0\rangle, \lambda x.\ \mathrm{first}(x)\rangle \\
P_c[\![\texttt{cdr}]\!] &= \langle\langle 0,0\rangle, \lambda x.\ \mathrm{second}(x)\rangle \\
P_c[\![\texttt{null}]\!] &= \langle\langle 0,0\rangle, \lambda x.\langle\langle 0,0\rangle, err\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
P_e[\![c]\!]env_p &= P_c[\![c]\!] \\
P_e[\![x]\!]env_p &= env_p[\![x]\!] \\
P_e[\![e_1 + e_2]\!]env_p &= \langle\langle 0,0\rangle, err\rangle \ /^*\ \text{same for } e_1 - e_2 \text{ and } e_1 = e_2\ ^*/ \\
P_e[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]env_p &= \text{if } \mathbf{Oracle}(e_1) \text{ then } (P_e[\![e_2]\!]env_p)_{(2)} \\
&\qquad\qquad \text{else } (P_e[\![e_3]\!]env_p) \\
P_e[\![e_1 e_2]\!]env_p &= (P_e[\![e_1]\!]env_p)_{(2)}\ (P_e[\![e_2]\!]env_p) \\
P_e[\![\texttt{lambda}(x).e]\!]env_p &= \langle V, \lambda y.P_e[\![e]\!]env_p[x \mapsto y]\rangle
\end{aligned}
$$

where

$$
V = \langle 0,0\rangle \sqcup \Big(\bigsqcup_{z \in F^{non-list}} (env_p[\![z]\!])_{(1)}\Big) \sqcup \Big(\bigsqcup_{z \in F^{list}}\ \Big(\bigsqcup_{p\ \mathbf{in}\ (env_p[\![z]\!])} p_{(1)}\Big)\Big)
$$

$p\ \mathbf{in}\ (env_p[\![z]\!])$ denotes that p is an refined escape pair in $env_p[\![z]\!]$,

$F^{non-list}$ = Set of non-list type free identifiers in $(\texttt{lambda}(x).e)$, and

F^{list} = Set of list type free identifiers in $(\texttt{lambda}(x).e)$.

$$
P_e[\![\texttt{letrec}\ x_1 = e_1; \ldots; x_n = e_n; \texttt{in}\ e]\!]env_p = P_e[\![e]\!]env'_p
$$

where $env'_p = env_p[x_1 \mapsto P_e[\![e_1]\!]env'_p, \ldots, x_n \mapsto P_e[\![e_n]\!]env'_p]$

$$
P_{pr}[\![pr]\!] = P_e[\![pr]\!]nullenv_p
$$

Figure 3.4: Refined Escape Semantic Functions

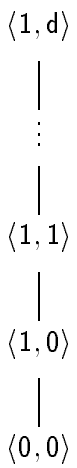

Figure 3.5: The Abstract Basic Refined Escape Domain

escape domain B_p as is shown in Figure 3.5. The ordering is also defined as follows:

$$\langle 0,0\rangle \sqsubseteq \langle 1,0\rangle \sqsubseteq \langle 1,1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,d-1\rangle \sqsubseteq \langle 1,d\rangle$$

But, the interpretation of elements of $\hat{B}_p$ is defined differently from the interpretation of B_p as follows:

- $\langle 1,i\rangle$: Only the bottom $\leq i$ spines of an interesting object *may be* contained in the value of the expression. (If an interesting object is not a list then i will always be 0, which means that an indivisible interesting object *may be* contained in the value of the expression.)
- $\langle 0,0\rangle$: *No* part of any interesting object is contained in the value of the expression.

The abstraction of the refined escape semantic subdomains for list type expressions is done by representing lists as finite objects, i.e. by combining the escape pairs of all its elements into a single escape pair. The abstract refined escape semantic domain $\hat{D}_p$ and the domain $\hat{E}_p$ of abstract refined escape environments are defined as follows:

$$\begin{aligned}
\hat{D}_p &= \sum_{\tau} \hat{D}_p^{\tau} \qquad \text{/* Abstract refined escape semantic domain */} \\
\hat{E}_p &= Id \rightarrow \hat{D}_p \quad \text{/* Domain of abstract refined escape environments */}
\end{aligned}$$

The abstract refined escape subdomain $\hat{D}_p^{\tau}$ for expressions of type τ is defined as follows:

$$
\begin{array}{lcll}
\hat{D}_p^{int} & = & \hat{B}_p \times \{err\} & \text{abstract subdomain for integers} \\
\hat{D}_p^{bool} & = & \hat{B}_p \times \{err\} & \text{abstract subdomain for booleans} \\
\hat{D}_p^{\tau_1 \rightarrow \tau_2} & = & \hat{B}_p \times (\hat{D}_p^{\tau_1} \rightarrow \hat{D}_p^{\tau_2}) & \text{abstract subdomain for functions of type } \tau_1 \rightarrow \tau_2 \\
\hat{D}_p^{\tau\ list} & = & \hat{D}_p^{\tau} & \text{abstract subdomain for lists of type } \tau\ list
\end{array}
$$

The abstract refined escape semantic functions are defined as follows:

$$
\begin{array}{lcll}
\hat{P}_c & : & Con \rightarrow \hat{D}_p & /^* \text{ Abstract refined escape function for constants } ^*/ \\
\hat{P}_e & : & Exp \rightarrow \hat{E}_p \rightarrow \hat{D}_p & /^* \text{ Abstract refined escape function for expressions } ^*/ \\
\hat{P}_{pr} & : & Program \rightarrow \hat{D}_p & /^* \text{ Abstract refined escape function for programs } ^*/
\end{array}
$$

The abstract refined escape semantic functions are given in Figure 3.6.

The abstract value of `cons` returns a single refined escape pair that is approximating a list of refined escape pairs by taking the least upper bound of its two arguments. The $\texttt{car}^{\texttt{s}}$ denotes a `car` that is applied to an argument of a list type with s spines. For each `car` in a program, `s` can be determined statically by type checking. It may be arbitrarily large, but is fixed at compile-time. The abstract value of `car` is defined as follows: $\texttt{car}^{\texttt{s}}$ takes a list with `s` spines as an argument, and returns a list with $(\texttt{s} - 1)$ spines when $\texttt{s} > 1$ or a non-list object when $\texttt{s} = 1$. In any case, the result cannot contain an interesting object with `s` spines. The abstract value of `cdr` just returns its argument list. $\hat{env}_p$ is any abstract refined escape environment in $\hat{E}_p$, and $\hat{nullenv}_p$ is an abstract refined escape environment that maps every identifier to the least element of its abstract refined escape semantic domain.

The safety of interpretation under the abstract refined escape semantics with respect to the exact refined escape semantics can be proved as follows. Let u and v be values of an expression e of type τ in the abstract refined escape domain $\hat{D}_p$ and the exact refined escape domain D_p, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that the abstract refined escape semantic value u is a *safe* approximation (with respect to refined escape information) of the exact refined escape semantic value v iff

$$
(\bigsqcup_{p \textbf{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k)} p_{(1)}) \sqsubseteq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)}
$$

for all $k \leq n$ where s_i is a *safe* approximation of t_i for all $i \leq k$.

Theorem 3.1 (Safety) *For any expression e, and environments env_p and $\hat{env}_p$ such that for all y, $\hat{env}_p[\![y]\!]$ is safe for $env_p[\![y]\!]$, $\hat{P}_e[\![e]\!]\hat{env}_p$ is safe (with respect to refined escape information) for $P_e[\![e]\!]env_p$.*

$$
\begin{array}{lcl}
\hat{P}_c[\![c]\!] & = & \langle\langle 0,0\rangle, err\rangle, c = \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\} \\
\hat{P}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (The bottom element in } \hat{D}_p^\tau) \\
\hat{P}_c[\![\texttt{cons}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\
\hat{P}_c[\![\texttt{car}^{\texttt{s}}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\mathrm{sub}^{\mathrm{s}}(x)\rangle \\
 & & \text{where} \\
 & & \mathrm{sub}^{\mathrm{s}}(z) = \text{if } (z_{(1)(2)} = \mathrm{s}) \text{ then } \langle\langle z_{(1)(1)}, \max(z_{(1)(2)} - 1, 0)\rangle, z_{(2)}\rangle \text{ else } z \\
\hat{P}_c[\![\texttt{cdr}]\!] & = & \langle\langle 0,0\rangle, \lambda x.x\rangle \\
\hat{P}_c[\![\texttt{null}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle\langle 0,0\rangle, err\rangle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\hat{P}_e[\![c]\!]e\hat{n}v_p & = & \hat{P}_c[\![c]\!] \\
\hat{P}_e[\![x]\!]e\hat{n}v_p & = & e\hat{n}v_p[\![x]\!] \\
\hat{P}_e[\![e_1 + e_2]\!]e\hat{n}v_p & = & \langle\langle 0,0\rangle, err\rangle \text{ /* same for } e_1 - e_2 \text{ and } e_1 = e_2 \text{ */} \\
\hat{P}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\hat{n}v_p & = & (\hat{P}_e[\![e_2]\!]e\hat{n}v_p)_{(2)} \sqcup (\hat{P}_e[\![e_3]\!]e\hat{n}v_p) \\
\hat{P}_e[\![e_1 e_2]\!]e\hat{n}v_p & = & (\hat{P}_e[\![e_1]\!]e\hat{n}v_p)_{(2)}\ (\hat{P}_e[\![e_2]\!]e\hat{n}v_p) \\
\hat{P}_e[\![\texttt{lambda}(x).e]\!]e\hat{n}v_p & = & \langle\hat{V}, \lambda y.\hat{P}_e[\![e]\!]e\hat{n}v_p[x \mapsto y]\rangle
\end{array}
$$

where
$\hat{V} = \langle 0,0\rangle(\bigsqcup_{z \in F}(env_p[\![z]\!])_{(1)})$ and
F = Set of free identifiers in $(\texttt{lambda}(x).e)$.

$$
\hat{P}_e[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n; \texttt{in } e]\!]e\hat{n}v_p = \hat{P}_e[\![e]\!]e\hat{n}v_p'
$$

where $e\hat{n}v_p' = e\hat{n}v_p[x_1 \mapsto \hat{P}_e[\![e_1]\!]e\hat{n}v_p', \ldots, x_n \mapsto \hat{P}_e[\![e_n]\!]e\hat{n}v_p']$

$$
\hat{P}_{pr}[\![pr]\!] = \hat{P}_e[\![pr]\!]nul\hat{l}env_p
$$

Figure 3.6: Abstract Refined Escape Semantic Functions

Proof : We can prove by structural induction on expression e.

I. Base Case:

1. $e = c$: $\hat{P}_e[\![c]\!]e\hat{n}v_p = \hat{P}_c[\![c]\!]$ and $P_e[\![c]\!]env_p = P_c[\![c]\!]$. For $c \in \{\ldots,$ `-1, 0, 1, ...,` `true, false, nil, null` $\}$, $P_c[\![c]\!] = \hat{P}_c[\![c]\!] = \langle\langle 0, 0\rangle, err\rangle$. It holds for $c =$ `cons`, because, if x_1 and x_2 are safe for y_1 and y_2 respectively, $x_1 \sqcup x_2$ is also safe for a list consisting of y_1 and y_2. For $c =$ `car`^{s}, it holds because if x is safe for y then sub$^s(x)$ is safe for first(y). It holds for $c =$ `cdr`, because $P_c[\![c]\!] = \hat{P}_c[\![c]\!] = \langle\langle 0, 0\rangle, \lambda x.x\rangle$.

2. $e = x$: $\hat{P}_e[\![x]\!]e\hat{n}v_p = e\hat{n}v_p[\![x]\!]$ and $P_e[\![x]\!]env_p = env_p[\![x]\!]$. Since, for all y, $e\hat{n}v_p[\![y]\!]$ is safe for $env_p[\![y]\!]$, it clearly holds.

II. Structural Induction Step: Assume that $\hat{P}_e[\![e]\!]e\hat{n}v_p$ is safe for $P_e[\![e]\!]env_p$ for expressions such as e_0, e_1, e_2, e_3 and e_n (structural induction hypothesis). Then, we show that $\hat{P}_e[\![e]\!]e\hat{n}v_p$ is safe for $P_e[\![e]\!]env_p$ for $e = e_1 + e_2$, `if` e_1 `then` e_2 `else` e_3, e_1e_2, `lambda`$(x).e_1$, and `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0. This step can be proved in an exactly same way to the safety proof of the abstract escape semantics with respect to the exact escape semantics. □

Theorem 3.2 (Termination) *For any (finite) program pr ∈ Program, $\hat{P}_{pr}[\![pr]\!]$ is computable.*

Proof : Every functional in the refined escape domain that is defined through the abstract refined escape semantic functions is composed of the operators such as the least upper bound operator $\sqcup$ and sub^s, which are monotonic. Furthermore, each subdomain $\hat{D}_p^\tau$ is finite. □

3.2.4 Refined Escapement Testing

The abstract refined escape semantics can be used as a basis for inferring refined escape information because interpretation under this semantics is effective. We describe the refined escape testing algorithms in which the abstract functions are used to detect the refined escape properties of the corresponding functions in a program.

Global Refined Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position i of an interesting parameter, and an abstract refined escape semantic environment $e\hat{n}v_p$ mapping f to an element of $\hat{D}_p$, the global refined escape test function G_rescape? determines how much of the i^{th} parameter of f could possibly escape f globally. It is defined as follows:

$$\mathsf{G_rescape?}(f, i, e\hat{n}v_p) = (\hat{P}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_p[x_i \mapsto y_i])_{(1)}$$

where

$$y_i = \langle\langle 1, s_i\rangle, W^{\tau_i}\rangle, \ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), for all $j \leq n$ and $j \neq i$,

$$y_j = \langle\langle 0, 0\rangle, W^{\tau_j}\rangle, \ \text{/* Other parameters are not interesting objects */}$$

and τ_i is the type of the i^{th} parameter of f. Note that the whole i^{th} argument with s_i spines is interesting, and any other argument is not interesting. Then, from the result of the global refined escape test function, we can conclude as follows:

- If $\mathsf{G_rescape?}(f, i, e\hat{n}v_p) = \langle 0, 0\rangle$ then we conclude that none of the i^{th} argument escapes f in any possible application of f to n arguments.
- If $\mathsf{G_rescape?}(f, i, e\hat{n}v_p) = \langle 1, k\rangle$ then we conclude that, if $s_i \geq 1$ then, the top $(s_i - k)$ spines of the i^{th} argument *do not* escape f in any possible application of f to n arguments, but the bottom k spines of the i^{th} argument *could* escape f in some application of f to n arguments. If $s_i = 0$ then the i^{th} argument, which is not a list type, *could* escape in some application of f to n arguments.

Local Refined Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in an application $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and an abstract escape semantic environment $e\hat{n}v_p$ mapping f and the free identifiers within e_1 through e_n to elements of $\hat{D}_p$, the local escape test function $\mathsf{L_rescape?}$ determines how much of the i^{th} parameter of f could escape f in the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$\mathsf{L_rescape?}(f, i, e_1, \ldots, e_n, e\hat{n}v_p) = (\hat{P}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_p[x_i \mapsto y_i])_{(1)}$$

where

$$y_i = \langle\langle 1, s_i\rangle, (\hat{P}_e[\![e_i]\!]\ e\hat{n}v_p)_{(2)}\rangle, \ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), and for all $j \leq n$ and $j \neq i$,

$$y_j = \langle\langle 0, 0\rangle, (\hat{E}_e[\![e_j]\!]\ e\hat{n}v_p)_{(2)}\rangle. \ \text{/* Other parameters are not interesting objects */}$$

Then, from the result of the local refined escape test function, we can conclude that:

- If L_rescape?$(f, i, e_1, \ldots, e_n, e\hat{n}v_p) = \langle 0, 0 \rangle$ then none of the i^{th} argument escapes f in the particular application of f to e_1 through e_n.
- If L_rescape?$(f, i, e_1, \ldots, e_n, e\hat{n}v_p) = \langle 1, k \rangle$ then we conclude that, if $s_i \geq 1$ then, the top $(s_i - k)$ spines of the i^{th} argument *do not* escape f in the particular application of f to e_1 through e_n, but the bottom k spines of the i^{th} argument *could* escape. If $s_i = 0$ then the i^{th} argument, which is not a list type, *could* escape in the particular application of f to e_1 through e_n.

Examples

As an example, consider the following partition sort program:

```
letrec ps x = if (null x) then nil
              else letrec y = split (car x) (cdr x) nil nil;
                   in append (ps (car y))
                             (cons (car x) (ps (car (cdr y))));

       split p x l h = if (null x) then (cons l (cons h nil))
                       elseif (car x)<p then
                              split p (cdr x) (cons (car x) l) h
                       else split p (cdr x) l (cons (car x) h);

       append x y = if (null x) then y
                    else cons (car x) (append (cdr x) y);
in ps [5,2,7,1,3,4]
```

We assume that the type of each function is given by

ps : $int\ list \rightarrow int\ list$
split : $int \rightarrow int\ list \rightarrow int\ list \rightarrow int\ list \rightarrow int\ list\ list$
append : $int\ list \rightarrow int\ list \rightarrow int\ list$

From type checking, each car in the program can be annotated as follows:

```
ps x = if (null x) then nil
       else letrec y = split (CAR1 x) (cdr x) nil nil;
            in append (ps (CAR2 y))
                      (cons (CAR1 x) (ps (CAR2 (cdr y))));
```

```
split p x l h = if (null x) then (cons l (cons h nil))
                elseif (CAR1 x)<p then
                       split p (cdr x) (cons (CAR1 x) l) h
                else split p (cdr x) l (cons (CAR1 x) h);

append x y = if (null x) then y
             else cons (CAR1 x) (append (cdr x) y);
```

where `CARi` denotes a `car` that takes as its argument a list with `i` spines. The refined escape semantic values *append*, *split*, and *ps* of `append`, `split`, and `ps` (shown uncurried for convenience) are expressed as follows:

$$
\begin{aligned}
&\mathit{append}\ x\ y \ = \ y \sqcup (\mathrm{sub}^1(x) \sqcup \mathit{append}\ x\ y)\\
&\mathit{split}\ p\ x\ l\ h \ = \ l \sqcup h \sqcup (\mathit{split}\ p\ x\ (\mathrm{sub}^1(x) \sqcup l)\ h) \sqcup (\mathit{split}\ p\ x\ \ (\mathrm{sub}^1(x) \sqcup h))\\
&\mathit{ps}\ x \ = \ \mathit{append}\ (\mathit{ps}\ \mathrm{sub}^2(\mathit{split}\ \mathrm{sub}^1(x)\ x\ \langle\langle 0,0\rangle \mathit{err}\rangle\ \langle\langle 0,0\rangle \mathit{err}\rangle))\\
&\qquad\quad (\mathrm{sub}^1(x) \sqcup (\mathit{ps}\ \mathrm{sub}^2(\mathit{split}\ \mathrm{sub}^1(x)\ x\ \langle\langle 0,0\rangle \mathit{err}\rangle\ \langle\langle 0,0\rangle \mathit{err}\rangle)))
\end{aligned}
$$

The meaning of each function in the refined escape semantic domain is found by fixpoint iteration. The fixpoint iteration for `append`:

$$
\begin{aligned}
\mathit{append}^{(0)}\ x\ y &= \bot_{\mathit{int\ list}}\\
\mathit{append}^{(1)}\ x\ y &= y \sqcup (\mathrm{sub}^1(x) \sqcup \mathit{append}^{(0)}\ x\ y)\\
&= y \sqcup \mathrm{sub}^1(x)\\
\mathit{append}^{(2)}\ x\ y &= y \sqcup (\mathrm{sub}^1(x) \sqcup \mathit{append}^{(1)}\ x\ y)\\
&= y \sqcup (\mathrm{sub}^1(x) \sqcup (y \sqcup \mathrm{sub}^1(x)))\\
&= y \sqcup \mathrm{sub}^1(x)
\end{aligned}
$$

Since $\mathit{append}^{(2)} = \mathit{append}^{(3)}$, we have that

$$\mathit{append} = \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.y \sqcup \mathrm{sub}^1(x)\rangle\rangle.$$

The fixpoint iteration for `split`:

$$
\begin{array}{lcl}
split^{(0)}\ p\ x\ l\ h & = & \bot_{(int\ list)\ list} \\
split^{(1)}\ p\ x\ l\ h & = & l \sqcup h \sqcup \bot_{(int\ list)\ list} \sqcup \bot_{(int\ list)\ list} \\
 & = & l \sqcup h \\
split^{(2)}\ p\ x\ l\ h & = & l \sqcup h \sqcup (split^{(1)}\ p\ x\ (\mathrm{sub}^1(x) \sqcup l)\ h) \sqcup (split^{(1)}\ p\ x\ \ (\mathrm{sub}^1(x) \sqcup h)) \\
 & = & l \sqcup h \sqcup ((\mathrm{sub}^1(x) \sqcup l) \sqcup h) \sqcup ((\mathrm{sub}^1(x) \sqcup h)) \\
 & = & l \sqcup h \sqcup \mathrm{sub}^1(x) \\
split^{(3)}\ p\ x\ l\ h & = & l \sqcup h \sqcup (split^{(2)}\ p\ x\ (\mathrm{sub}^1(x) \sqcup l)\ h) \sqcup (split^{(2)}\ p\ x\ \ (\mathrm{sub}^1(x) \sqcup h)) \\
 & = & l \sqcup h \sqcup ((\mathrm{sub}^1(x) \sqcup l) \sqcup h) \sqcup (l \sqcup (\mathrm{sub}^1(x) \sqcup h) \sqcup \mathrm{sub}^1(x)) \\
 & = & l \sqcup h \sqcup \mathrm{sub}^1(x)
\end{array}
$$

Since $split^{(2)} = split^{(3)}$, we have that

$$split = \langle\langle 0,0\rangle, \lambda p.\langle p_{(1)}, \lambda x.\langle p_{(1)} \sqcup x_{(1)}, \lambda l.\langle p_{(1)} \sqcup x_{(1)} \sqcup l_{(1)}, \lambda h.l \sqcup h \sqcup \mathrm{sub}^1(x)\rangle\rangle\rangle\rangle.$$

The fixpoint iteration for `ps`:

$$
\begin{array}{lcl}
ps^{(0)}\ x & = & \bot_{int\ list} \\
ps^{(1)}\ x & = & append(ps^{(0)}\ \mathrm{sub}^2(\mathrm{sub}^1(x)))\ (\mathrm{sub}^1(x) \sqcup (ps^{(0)}\ \mathrm{sub}^2(\mathrm{sub}^1(x)))) \\
 & = & append\ \bot_{int\ list}\ (\mathrm{sub}^1(\mathrm{sub}^1(x)) \sqcup \bot_{int\ list}) \\
 & = & \mathrm{sub}^1(x) \\
ps^{(2)}\ x & = & append(ps^{(1)}\ \mathrm{sub}^2(\mathrm{sub}^1(x)))\ (\mathrm{sub}^1(x) \sqcup (ps^{(1)}\ \mathrm{sub}^2(\mathrm{sub}^1(x)))) \\
 & = & append\ \mathrm{sub}^1(\mathrm{sub}^2(\mathrm{sub}^1(x)))\ (\mathrm{sub}^1(x) \sqcup \mathrm{sub}^1(\mathrm{sub}^2(\mathrm{sub}^1(x)))) \\
 & = & \mathrm{sub}^1(x)
\end{array}
$$

Since $ps^{(1)} = ps^{(2)}$, we have that

$$ps = \langle\langle 0,0\rangle, \lambda x.\mathrm{sub}^1(x)\rangle.$$

Let $e\hat{n}v_p$ be defined as $e\hat{n}v_p = [\texttt{append} \mapsto append, \texttt{split} \mapsto split, \texttt{ps} \mapsto ps]$. Then,

$$
\begin{array}{lcl}
\texttt{G_rescape?}(\texttt{append}, 1, e\hat{n}v_p) & = & (\hat{P}_e[\![\texttt{append x y}]\!]\hat{env'_p})_{(1)} \\
 & = & (\langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,0\rangle
\end{array}
$$

where $\hat{env'_p} = e\hat{n}v_p[\texttt{x} \mapsto \langle\langle 1,1\rangle, err\rangle, \texttt{y} \mapsto \langle\langle 0,0\rangle, err\rangle]$.

$$
\begin{array}{lcl}
\texttt{G_rescape?}(\texttt{append}, 2, e\hat{n}v_p) & = & (\hat{P}_e[\![\texttt{append x y}]\!]\hat{env'_p})_{(1)} \\
 & = & (\langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,1\rangle
\end{array}
$$

where $\hat{env'_p} = e\hat{n}v_p[\texttt{x} \mapsto \langle\langle 0,0\rangle, err\rangle, \texttt{y} \mapsto \langle\langle 1,1\rangle, err\rangle]$. Thus, we can conclude that `append` returns all of its second argument `y`, and all but the top spine of the first argument `x`.

$$\begin{aligned}
\mathsf{G_rescape?}(\mathtt{split}, 1, e\hat{n}v_p) &= (\hat{P}_e[\![\mathtt{split\ p\ x\ l\ h}]\!]e\hat{n}v'_p)_{(1)} \\
&= (\langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
&= \langle 0,0\rangle
\end{aligned}$$

where $e\hat{n}v'_p = e\hat{n}v_p[\mathtt{p} \mapsto \langle\langle 1,0\rangle, err\rangle, \mathtt{x}, \mathtt{l}, \mathtt{h} \mapsto \langle\langle 0,0\rangle, err\rangle]$.

$$\begin{aligned}
\mathsf{G_rescape?}(\mathtt{split}, 2, e\hat{n}v_p) &= (\hat{P}_e[\![\mathtt{split\ p\ x\ l\ h}]\!]e\hat{n}v'_p)_{(1)} \\
&= (\langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
&= \langle 1,0\rangle
\end{aligned}$$

where $e\hat{n}v'_p = e\hat{n}v_p[\mathtt{x} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{p}, \mathtt{l}, \mathtt{h} \mapsto \langle\langle 0,0\rangle, err\rangle]$.

$$\begin{aligned}
\mathsf{G_rescape?}(\mathtt{split}, 3, e\hat{n}v_p) &= (\hat{P}_e[\![\mathtt{split\ p\ x\ l\ h}]\!]e\hat{n}v'_p)_{(1)} \\
&= (\langle\langle 1,1\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
&= \langle 1,1\rangle
\end{aligned}$$

where $e\hat{n}v'_p = e\hat{n}v_p[\mathtt{l} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{p}, \mathtt{x}, \mathtt{h} \mapsto \langle\langle 0,0\rangle, err\rangle]$.

$$\begin{aligned}
\mathsf{G_rescape?}(\mathtt{split}, 4, e\hat{n}v_p) &= (\hat{P}_e[\![\mathtt{split\ p\ x\ l\ h}]\!]e\hat{n}v'_p)_{(1)} \\
&= (\langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 1,1\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
&= \langle 1,1\rangle
\end{aligned}$$

where $e\hat{n}v'_p = e\hat{n}v_p[\mathtt{h} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{p}, \mathtt{x}, \mathtt{l} \mapsto \langle\langle 0,0\rangle, err\rangle]$. Thus, we can conclude that `split` returns all of its third and fourth arguments `l` and `h`, none of the first argument `p`, and all but the top spine of the second argument `x`.

$$\begin{aligned}
\mathsf{G_rescape?}(\mathtt{ps}, 1, e\hat{n}v_p) &= (\hat{P}_e[\![\mathtt{ps\ \ x}]\!]\ e\hat{n}v_p[\mathtt{x} \mapsto \langle\langle 1,1\rangle, err\rangle])_{(1)} \\
&= (\mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
&= \langle 1,0\rangle
\end{aligned}$$

Thus, we conclude that `ps` returns all but the top spine of its argument `x`.

3.3 Improving Precision of Refined Escapement

We describe an improved abstraction of the exact refined escape semantics achieved by extending the basic abstract refined escape domain to include additional information about the positions of objects in a list. For each expression, its corresponding value in the improved abstract refined escape semantic domain tells us how much of an interesting object may be contained at some position in the value of the expression ("maybe partial-escape and where"). The basic improved abstract escape domain, $\hat{B}_p$, for some fixed d and p is

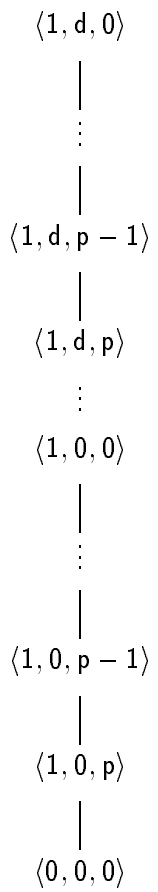

Figure 3.7: The Improved Abstract Basic Refined Escape Domain

a $((d+1)(p+1)+1)$-element domain of triples as shown in Figure 3.7. The ordering on triples in $\hat{B}_p$ is defined as follows:

$$\langle 0,0,0\rangle \sqsubseteq \langle 1,0,p\rangle \sqsubseteq \langle 1,0,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,0,0\rangle \ldots \langle 1,d,p\rangle \sqsubseteq \langle 1,d,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,d,0\rangle$$

The interpretation of the elements of $\hat{B}_p$ is defined as follows:

- $\langle 1,i,k\rangle$: Only the bottom $\leq i$ spines of an interesting object *may be* contained only at positions $\geq k$ in the value of the expression, if that value is a list. (If an interesting object is not a list then i will always be 0, which means that an indivisible interesting object *may be* contained in the value of the expression.)

- $\langle 0,0,0\rangle$: *No* part of any interesting object *is* contained in the value of the expression.

The improved abstract refined escape semantic domain $\tilde{D}_p$ and the domain $\tilde{E}_p$ of improved abstract refined escape environments are defined as follows:

$$\begin{array}{lcll} \tilde{D}_p & = & \sum_{\tau} \tilde{D}_p^{\tau} & \text{/* Improved abstract refined escape semantic domain */} \\ \tilde{E}_p & = & Id \rightarrow \tilde{D}_p & \text{/* Domain of improved refined escape environments */} \end{array}$$

The improved abstract refined escape subdomains $\tilde{D}_p^{\tau}$ for expressions of type τ are defined as follows:

$$\begin{array}{lcll} \tilde{D}_p^{int} & = & \hat{B}_p \times \{err\} & \text{improved abstract subdomain for integers} \\ \tilde{D}_p^{bool} & = & \hat{B}_p \times \{err\} & \text{improved abstract subdomain for booleans} \\ \tilde{D}_p^{\tau_1 \rightarrow \tau_2} & = & \hat{B}_p \times (\tilde{D}_p^{\tau_1} \rightarrow \tilde{D}_p^{\tau_2}) & \text{improved abstract subdomain for functions} \\ \tilde{D}_p^{\tau\ list} & = & \tilde{D}_p^{\tau} & \text{improved abstract subdomain for lists} \end{array}$$

The improved abstract refined escape semantic functions are as follows:

$$\begin{array}{lcll} \tilde{P}_c & : & Con \rightarrow \tilde{D}_p & \text{/* Improved semantic function for constants */} \\ \tilde{P}_e & : & Exp \rightarrow \tilde{E}_p \rightarrow \tilde{D}_p & \text{/* Improved semantic function for expressions */} \\ \tilde{P}_{pr} & : & Program \rightarrow \tilde{D}_p & \text{/* Improved semantic function for programs */} \end{array}$$

The improved abstract refined escape semantic function $\tilde{P}_c : Con \rightarrow \tilde{D}_p$ that gives refined escape meaning to constants is given in Figure 3.8. The improved abstract value of `cons` takes two arguments, and returns an improved refined escape pair of their least upper bound by updating the position information of an interesting object in the result list. The improved abstract value of $\texttt{car}^{\texttt{s}}$ returns its argument according to the position of an interesting object. $\texttt{car}^{\texttt{s}}$ takes a list with `s` spines and returns the element that was in the 0^{th} position and has $(\texttt{s}-1)$ spines. Thus, the result cannot contain an interesting object with `s` spines or

$$
\begin{array}{lcl}
\tilde{P}_c[\![c]\!] & = & \langle\langle 0,0,0\rangle, err\rangle,\ \ c\{\ldots,\texttt{-1},\texttt{0},\texttt{1},\ldots,\texttt{true},\texttt{false}\} \\
\tilde{P}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (The bottom element in } \tilde{D}_p^\tau) \\
\tilde{P}_c[\![\texttt{cons}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\mathrm{rpush}(x,y)\rangle\rangle \\
 & & \text{where} \\
 & & \mathrm{rpush}(x,y) = \ \text{if } (x_{(1)(1)} = 1) \\
 & & \qquad \text{then } \langle\langle 1, x_{(1)(2)} \sqcup y_{(1)(2)}, 0\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
 & & \qquad \text{elseif } (y_{(1)(1)} = 1) \\
 & & \qquad \text{then } \langle\langle 1, y_{(1)(2)}, \min(y_{(1)(3)}+1, p)\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
 & & \qquad \text{else } \langle\langle 0,0,0\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
\tilde{P}_c[\![\texttt{car}^{\texttt{s}}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\mathrm{rsub}^{\mathrm{s}}(x)\rangle \\
 & & \text{where} \\
 & & \mathrm{rsub}^{\mathrm{s}}(z) = \ \text{if } (z_{(1)(1)} = 0) \text{ or } (z_{(1)(3)} > 0) \text{ then } \langle\langle 0,0,0\rangle, z_{(2)}\rangle \\
 & & \qquad \text{else } \langle\langle \mathrm{dec}^{\mathrm{s}}(z_{(1,2)}), 0\rangle, z_{(2)}\rangle \\
 & & \mathrm{dec}^{\mathrm{s}}(y) = \ \text{if } (y_{(1)} - 1 = s) \text{ then } \langle \max(y_{(1)} - 1, 0), y_{(2)}\rangle \\
 & & \qquad \text{else } y \\
\tilde{P}_c[\![\texttt{cdr}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\mathrm{rrest}(x)\rangle \\
 & & \text{where} \\
 & & \mathrm{rrest}(z) = \ \langle\langle z_{(1)(1)}, z_{(1)(2)}, \max[z_{(1)(3)} - 1, 0]\rangle, z_{(2)}\rangle \\
\tilde{P}_c[\![\texttt{null}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\langle\langle 0,0,0\rangle, err\rangle
\end{array}
$$

Figure 3.8: Improved Abstract Refined Escape Semantic Functions

that occurred at a position ≥ 1 in the original list. The improved abstract value of `cdr` also updates the position information appropriately. Note that the improved abstract refined escape semantic function $\tilde{P}_c$ for `cons`, `car` and `cdr` provides more precise escape information than the abstract refined escape semantic function $\hat{P}_c$.

The improved abstract refined escape semantic function $\tilde{P}_e : Exp \rightarrow \tilde{E}_p \rightarrow \tilde{D}_p$ that gives refined escape meaning to expressions, and the improved abstract refined escape semantic function $\tilde{P}_{pr} : Program \rightarrow \tilde{D}_p$ that gives refined escape meaning to programs are defined similarly to the abstract refined escape semantic function $\hat{P}_e$ and $\hat{P}_{pr}$ respectively.

The safety of interpretation under the improved abstract refined escape semantics with respect to the exact refined escape semantics can be proved as follows. Let u and v be values of an expression e of type τ in $\tilde{D}_p$ and D_p, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that the improved abstract refined escape semantic value u is a *safe* approximation (with respect to refined escape information) for the exact refined escape semantic value v iff

$$\left(\bigsqcup_{p \textbf{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k)} p_{(1)} \right) \sqsubseteq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(1,2)}$$

and

$$(MIN_{p \textbf{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k) \& p_{(1)(1)}=1} \text{Position of } p) \geq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(3)}$$

for all $k \leq n$ where s_i is a *safe* approximation for t_i for all $i \leq k$.

Theorem 3.3 (Safety) *For any expression e, and environments env_p and $e\tilde{n}v_p$ such that for all y, $e\hat{n}v_p[\![y]\!]$ is safe for $env_p[\![y]\!]$, $\tilde{P}_e[\![e]\!]e\tilde{n}v_p$ is safe (with respect to refined escape information) for $P_e[\![e]\!]env_p$.*

Proof : We can prove by structural induction on expression e.

I. Base Case:

1. $e = c$: $\tilde{P}_e[\![c]\!]e\tilde{n}v_p = \tilde{P}_c[\![c]\!]$ and $P_e[\![c]\!]env_p = P_c[\![c]\!]$. For $c \in \{\ldots,$ `-1,0,1, ..., true, false, nil, null` $\}$, $P_c[\![c]\!] = \tilde{P}_c[\![c]\!] = \langle\langle 0,0,0\rangle, err\rangle$. It holds for $c =$ `car`, because if x_1 and x_2 are safe for y_1 and y_2 respectively then $\mathrm{rpush}(x, y)$ is also safe for a list of y_1 and y_2. For $c =$ `car`^{s}, it holds because if x is safe for y then then $\mathrm{rsub}^s(x)$ is also safe for $\mathrm{first}(y)$. It holds for $c =$ `cdr`, because if x is safe for y then then $\mathrm{rrest}(x)$ is also safe for $\mathrm{second}(y)$.

2. $e = x$: $\tilde{P}_e[\![x]\!]e\tilde{n}v_p = e\tilde{n}v_p[\![x]\!]$ and $P_e[\![x]\!]env_p = env_p[\![x]\!]$. Since, for all y, $e\tilde{n}v_p[\![y]\!]$ is safe for $env_p[\![y]\!]$, it clearly holds.

II. Structural Induction Step: Assume that $\tilde{P}_e[\![e]\!]e\tilde{n}v_p$ is safe for $P_e[\![e]\!]env_p$ for expressions

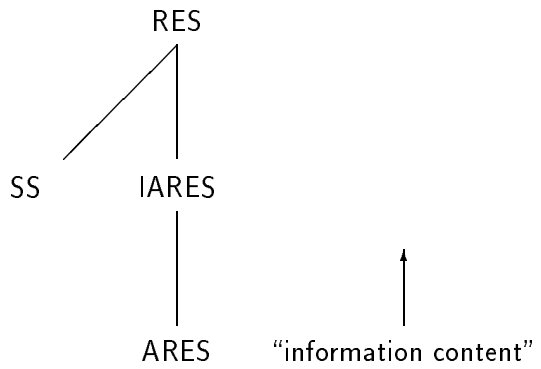

Figure 3.9: Relationship among Standard and Refined Escape Semantics

such as e_0, e_1, e_2, e_3 and e_n (structural induction hypothesis). Then, we show that $\tilde{P}_e[\![e]\!]e\tilde{n}v_p$ is safe for $P_e[\![e]\!]envp$ for $e = e_1 + e_2$, `if` e_1 `then` e_2 `else` e_3, e_1e_2, `lambda`$(x).e_1$, and `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0. This can be proved in an exactly the same as the proof of safety of the abstract refined escape semantics. □

Theorem 3.4 (Termination) *For any (finite) program* $pr \in Program$, $\tilde{P}_{pr}[\![pr]\!]$ *is computable.*

Proof : Every functional in the improved escape domain that is defined through the abstract improved refined escape semantic functions is composed of the operators such as the least upper bound operator $\sqcup$, rpush, rsub^s, and rrest. rpush, rsub^s and rrest are all monotonic operators. Also, each subdomain $\tilde{D}_p^\tau$ is finite. □

Theorem 3.5 (Precision Improvement) *The abstract refined escape semantics is equivalent (with respect to refined escape information) to the improved abstract refined escape semantics with* $p = 0$. *Thus, the improved abstract refined escape semantics with* $p > 0$ *provides more precise escape information than the abstract refined escape semantics.*

Proof : This can be proved in a similar way to the proof of Theorem 2.5. □

The relationships among the standard semantics (SS), the exact refined escape semantics (RES), the abstract refined escape semantics (ARES) and the improved abstract refined escape semantics (IARES) are shown in Figure 3.9

Improved Global Refined Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position i of an interesting parameter, and an improved abstract refined escape semantic environment $e\tilde{n}v_p$ mapping f to an element of $\tilde{D}_p$, the global improved refined escape test function G_irescape? which determines how much of the i^{th} parameter of f could possibly escape f globally is defined as follows:

$$\mathsf{G_irescape?}(f, i, e\tilde{n}v_p) = (\tilde{P}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\tilde{n}v_p[x_i \mapsto y_i])_{(1)(1,2)}$$

where

$$y_i = \langle\langle 1, s_i, 0\rangle, W^{\tau_i}\rangle,\ \text{/* The } i^{th} \text{ parameter is an interesting object */}$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), for all $j \leq n$ and $j \neq i$,

$$y_j = \langle\langle 0, 0, 0\rangle, W^{\tau_j}\rangle,\ \text{/* Other parameters are not interesting objects */}$$

and τ_i is the type of the i^{th} parameter of f. The result of the global improved refined escape test function is interpreted as follows:

- If $\mathsf{G_irescape?}(f, i, e\tilde{n}v_p) = \langle 0, 0\rangle$ then we can conclude that in any possible application of f to n arguments, none of the i^{th} argument escapes f.
- If $\mathsf{G_irescape?}(f, i, e\tilde{n}v_p) = \langle 1, k\rangle$ then we can conclude that if $s_i \geq 1$ then, in any possible application of f to n arguments, the top $(s_i - k)$ spines of the i^{th} argument *do not* escape f, but, in some application of f to n arguments, the bottom k spines of the i^{th} argument *could* escape f. If $s_i = 0$ then in some application of f to n arguments, the i^{th} argument, which is not a list type, *could* escape.

Improved Local Refined Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in a application context $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and an improved abstract escape semantic environment $e\tilde{n}v_p$ mapping f and the free identifiers within e_1 through e_n to elements of $\tilde{D}_p$, the local escape test function L_irescape? which determines how much of the i^{th} parameter of f could escape f in the evaluation of $f\ e_1\ \ldots\ e_n$ is defined as follows:

$$\mathsf{L_irescape?}(f, i, e_1, \ldots, e_n, e\tilde{n}v_p) = (\tilde{P}_e[\![f\ x_1\ \ldots\ x_n]\!]\ e\tilde{n}v_p[x_i \mapsto y_i])_{(1)(1,2)}$$

where

$y_i = \langle\langle 1, s_i, 0\rangle, (\tilde{P}_e[\![e_i]\!]\ e\tilde{n}v_p)_{(2)}\rangle$, /* The i^{th} parameter is an interesting object */

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), and for all $j \leq n$ and $j \neq i$,

$y_j = \langle\langle 0, 0, 0\rangle, (\tilde{E}_e[\![e_j]\!]\ e\tilde{n}v_p)_{(2)}\rangle$. /* Other parameters are not interesting objects */

The result of the local improved refined escape test function is interpreted as follows:

- If L_irescape?$(f, i, e_1, \ldots, e_n, e\tilde{n}v_p) = \langle 0, 0\rangle$ then we can conclude that in the particular application of f to e_1 through e_n, none of the i^{th} argument escapes f.
- If L_irescape?$(f, i, e_1, \ldots, e_n, e\tilde{n}v_p) = \langle 1, k\rangle$ then we can conclude that if $s_i \geq 1$ then, in the particular application of f to e_1 through e_n, the top $(s_i - k)$ spines of the i^{th} argument *do not* escape f, but the bottom k spines *could* escape f. If $s_i = 0$ then the i^{th} argument, which is not a list type, *could* escape in the particular application of f to e_1 through e_n.

3.4 Comparison to Escape Analysis and Complexity

The refined escape semantics, the abstract refined escape semantics, and the improved abstract refined escape semantics with some $d > 0$ provide more refined escape information than the escape semantics, the abstract escape semantics, and the improved abstract escape semantics, respectively. The relationship among the standard semantics, the escape semantics, the improved abstract escape semantics, the abstract escape semantics, the refined escape semantics, the improved abstract refined escape semantics, and the abstract refined escape semantics is shown in Figure 3.10.

The abstract interpretation framework for the refined escape analysis is the same as that for the escape analysis presented in the previous chapter except for the size of the abstract basic domain. While the escape analysis uses a two-element domain, the refined escape analysis uses a $(d+2)$-element domain where d is fixed and greater than 2. Thus, the order of worst-case time complexity of refined escape analysis is higher than but is comparable to that of escape analysis, which is exponential.

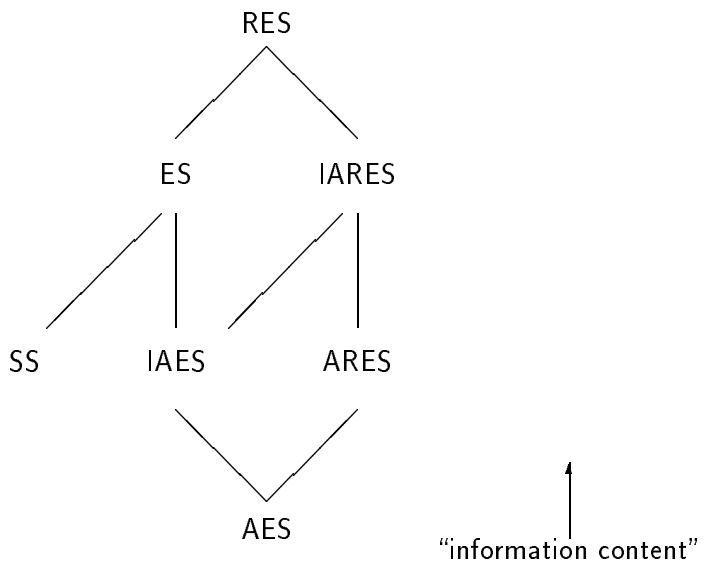

Figure 3.10: Relationship among Escape and Refined Escape Semantics

Chapter 4

Reference Escape Analysis

In reference counting schemes for automatic storage reclamation, each time a reference to an object is created or destroyed, the reference count of the object needs to be updated. This may involve expensive inter-processor message exchanges in distributed environments. In higher-order functional languages, exact information about the lifetime of a dynamically created reference to a heap-allocated object is generally unknown at compile-time. Thus, updating on the reference count of the object is performed whenever a reference to the object is created and destroyed. Such information, if inferred at compile-time, could be useful for improving the reference counting scheme for both uniprocessor and multiprocessor environments.

In this chapter, we present a method for computing, at compile-time, safe information about the relative lifetimes of dynamically created references to objects, such as arguments and local objects defined within a function, with respect to the lifetime of the function call. As before, the language we consider is a higher-order, monomorphic, and strict functional language. This method is based on a compile-time semantic analysis called *reference escape analysis*. First, we introduce a non-standard denotational semantics called *reference escape semantics* that describes the actual reference escape behavior, but is incomputable at compile time. An abstraction method of approximating the exact reference escape semantics which is safe with respect to the exact reference escape semantics and is computable at compile-time is then presented. Based on the *abstract reference escape semantics* and function transformation, we describe the reference escape testing algorithms which determine reference escape information. Another safe and computable abstraction of the exact reference escape semantics called *improved abstract reference escape semantics* that improves the precision of reference escape information using the position information of objects in a

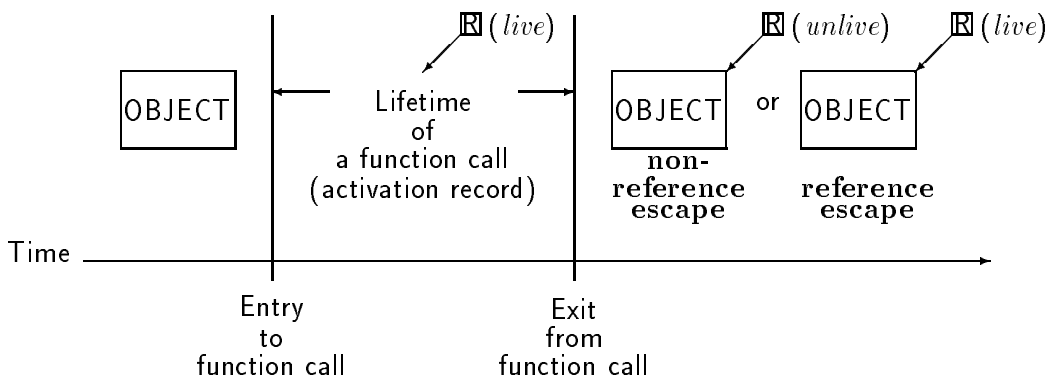

Figure 4.1: Escapement of References

list structure is also presented. Finally, the complexity of the reference escape analysis is discussed.

4.1 Escapement of References

An object needed in more than one place during a program's execution is typically handled in one of two ways: either the object itself is copied or a pointer to the object is copied. We describe the operational model in which references are created and destroyed. We choose a model that is commonly implemented in LISP and functional language systems, that of a call by value language with pointer semantics for heap allocated structures. References are created in three ways:

1. When a heap allocated object is created, a reference to that object is created and returned by the allocation procedure (e.g. `cons`).

2. When a heap allocated object is passed as a parameter in a function call, a reference to the object is copied into the activation record of the called function.

3. When an assignment occurs (in a `letrec`, for example), and the value of the right hand side is a heap-allocated object, a reference to the object is copied into the variable (or record field) on the left hand side.

Consider the following function definition:

```
f x y =  letrec g a b = cons a b
```

```
in cons x (g x y)
```

When `f` is called, its activation record contains two references to lists, corresponding to the parameters `x` and `y`. Thus, when `(g x y)` is evaluated, the references corresponding to `x` and `y` are copied into the activation record for `g`. Likewise, any argument to `cons` that is represented by a reference is also copied.

We analyze the lifetimes of a reference by determining its *escapement*, that is, whether or not a reference is returned out of the scope in which it was created. When does a reference escape? Intuitively, a reference can escape when it is placed in a structure, or closure that escapes. If a reference does not escape the scope of its creation, no reference counting operations are necessary when the reference is created or destroyed. In the above example, when the references corresponding to `x` and `y` are copied into `g`'s activation record, no reference count increment operation is required. Likewise, when `g` returns, no decrement operation is required. However, when a cons cell, corresponding to `(cons a b)`, is created, the reference counts of the objects pointed to by `a` and `b` must be incremented. This is because the lifetime of the cons cell exceeds that of `g` and `f`, and it cannot be determined at compile time, when the references contained in the cons cell will be destroyed.

We formally define the notion of escapement of references, and illustrate it in Figure 4.1.

Definition 4.1 (Global/Local Reference Escapement) Given a function f with n formal parameters and m locally defined objects, the j^{th} occurrence of the i^{th} parameter or locally defined object is said to

- *reference-escape* the function call to f *globally* if, in *some* possible application of f to n arguments, the reference associated with x_i outlives the the function call (by being contained in the result of the function application).
- *reference-escape* the function call to f *locally* in $(f\ e_1\ \ldots\ e_n)$ if, in the particular function application of $(f\ e_1\ \ldots\ e_n)$, the reference associated with x_i outlives the activation of the function call.

From the escapement of a reference with respect to its defining function, we can deduce its lifetime: If the reference associated with an occurrence of a parameter or local object does not escape the function call globally then we can conclude that the lifetime of the reference is confined to the lifetime of any possible call to the function. Similarly, if it does not reference-escape the function call locally in a particular function application then we can conclude that the lifetime of the reference is confined to the lifetime of that particular function call.

We will develop a general method for higher-order, monomorphic, strict functional languages to answer the following questions at compile-time:

- Given a function and an occurrence of a parameter or local object, does the reference associated with the occurrence reference-escape the function call globally ?
- Given a function in a particular application context and an occurrence of a parameter or local object that is defined within the function, does the reference associated with the occurrence reference-escape that function call ?

4.2 Function Transformation

In order to make each occurrence of each parameter of a function distinct, we introduce an auxiliary function f' for each function f. Then, we perform reference escape testing on f' to determine reference escape property of each occurrence of each parameter of f. Given a function

$$f\ x_1\ \ldots\ x_n\ =\ e,$$

the auxiliary function f' is given as follows:

$$f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots\ x_{n1}\ \ldots\ x_{no(n)}\ =\ e'$$

where $o(i)$ is the number of occurrences of x_i in e and e' is derived from e by replacing the j^{th} occurrence of x_i by x_{ij} for all i and j. Note that each parameter of f' will now have only one occurrence, and f' will be never called from anywhere and thus is not recursive. To determine the escape behavior of references associated with occurrences of parameters of f, we perform the test on its auxiliary function f'. For example, consider the following function.

```
f x y =  letrec g a b = cons a b;
         in cons x (g x y)
```

An auxiliary function `f'` derived from `f` is given as follows:

```
f' x1 x2 y =  letrec g a b = cons a b;
              in cons x1 (g x2 y)
```

Figure 4.2: The Basic Reference Escape Domain

4.3 Exact Reference Escape Semantics

We introduce an exact but uncomputable non-standard denotational semantics, called *reference escape semantics*, which describes the complete escaping behaviors of references for functions in a program. The reference associated with each occurrence of a parameter or local object will be analyzed separately to determine its reference-escape behavior. We say that a reference is *interesting* if it is the one whose escape behavior we are trying to determine. Thus, our reference escape semantics is defined in terms of a single interesting reference.

Representing Reference Escape Information

The meaning we will attach to the syntax of our language is information about escaping references. For each expression, we want its corresponding value in the reference escape semantic domain to be able to tell us if an interesting reference is not contained in the value of the expression ("non-reference-escape"), or if an interesting reference is contained in the value of the expression ("reference-escape"). Under the non-standard reference escape semantics, we represent the meaning of an expression as a pair, called *a reference escape pair*,

1. whose first element denotes the containment of an interesting reference in the value of the expression, and

2. whose second element denotes the functional behavior of the expression defined over the reference escape domain when the expression itself is applied to another expression.

For a non-list type expression, the corresponding value in the non-standard reference escape semantic domain D_r has two components; The first component is an element of a domain called a *basic reference escape domain*, B_r which is a two element domain showned Fig-

ure 4.2. The ordering is defined as $0 \sqsubseteq 1$. The interpretation of elements of B_r is defined as follows:

- 1 : An interesting reference *is* contained in the value of the expression.
- 0 : *No* interesting reference *is* contained in the value of the expression.

For expressions which have no higher-order behavior, err which is a function that can never be applied, is used. For a list type expression, the corresponding value in the reference escape semantic domain is a list that consist of corresponding reference escape values of its components.

Reference Escape Semantic Domains

The reference escape semantic domain D_r and the domain E_r of reference escape environments are defined as follows:

$$\begin{aligned} D_r &= \sum_{\tau} D_r^{\tau} && \text{/* Reference escape semantic domain */} \\ E_r &= Id \rightarrow D_r && \text{/* Domain of reference escape environments */} \end{aligned}$$

The reference escape domain D_r is a sum domain consisting of each subdomain for each type. The reference escape subdomain D_r^{τ} for expressions of type τ is defined as follows:

$$\begin{aligned} D_r^{int} &= B_r \times \{err\} && \text{subdomain for integers} \\ D_r^{bool} &= B_r \times \{err\} && \text{subdomain for booleans} \\ D_r^{\tau_1 \rightarrow \tau_2} &= B_r \times (D_r^{\tau_1} \rightarrow D_r^{\tau_2}) && \text{subdomain for functions of type } \tau_1 \rightarrow \tau_2 \\ D_r^{\tau\ list} &= (B_r \times \{err\}) + (D_r^{\tau} \times D_r^{\tau\ list}) && \text{subdomain for lists of type } \tau\ list \end{aligned}$$

Reference Escape Semantic Functions

The non-standard reference escape semantic functions are defined as follows:

$$\begin{aligned} R_c &: Con \rightarrow D_r && \text{/* Reference escape semantic function for constants */} \\ R_e &: Exp \rightarrow E_r \rightarrow D_r && \text{/* Reference escape semantic function for expressions */} \\ R_{pr} &: Program \rightarrow D_r && \text{/* Reference escape semantic function for programs */} \end{aligned}$$

The semantic equations for the reference escape semantic functions are expressed in Figure 4.3. Note that **Oracle** is used to resolve the exact behavior of the conditional expression `if`, which relies on the standard semantics. env_r is any exact reference escape environment in E_r, and $nullenv_r$ is a reference escape environment that maps every identifier on to the least element of its reference escape domain.

$$
\begin{aligned}
R_c[\![c]\!] &= \langle 0, err\rangle,\ \ c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}, \texttt{nil}^{\tau\ list}\} \\
R_c[\![\texttt{cons}]\!] &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\ \mathrm{pair}(x, y)\rangle\rangle \\
R_c[\![\texttt{car}]\!] &= \langle 0, \lambda x.\ \mathrm{first}(x)\rangle \\
R_c[\![\texttt{cdr}]\!] &= \langle 0, \lambda x.\ \mathrm{second}(x)\rangle \\
R_c[\![\texttt{null}]\!] &= \langle 0, \lambda x.\langle 0, err\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
R_e[\![c]\!]env_r &= R_c[\![c]\!] \\
R_e[\![x]\!]env_r &= env_r[\![x]\!] \\
R_e[\![e_1 + e_2]\!]env_r &= \langle 0, err\rangle \ /^*\ \text{same for } e_1 - e_2 \text{ and } e_1 = e_2\ ^*/ \\
R_e[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]env_r &= \text{if } \mathbf{Oracle}(e_1) \text{ then } (R_e[\![e_2]\!]env_r) \\
&\qquad\qquad \text{else } (R_e[\![e_2]\!]env_r) \\
R_e[\![e_1 e_2]\!]env_r &= (R_e[\![e_1]\!]env_r)_{(2)}\ (R_e[\![e_2]\!]env_r) \\
R_e[\![\texttt{lambda}(x).e]\!]env_r &= \langle V, \lambda y.R_e[\![e]\!]env_r[x \mapsto y]\rangle
\end{aligned}
$$

where

$V = \langle 0, 0\rangle \sqcup (\bigsqcup_{z \in F^{non-list}} (env_r[\![z]\!])_{(1)}) \sqcup (\bigsqcup_{z \in F^{list}} (\bigsqcup_{p\ \mathbf{in}\ env_r[\![z]\!]} p_{(1)}))$,
$p\ \mathbf{in}\ env_r[\![z]\!]$ denotes that p is an reference escape pair in $env_r[\![z]\!]$,
$F^{non-list}$ = Set of non-list type free identifiers in $(\texttt{lambda}(x).e)$, and
F^{list} = Set of list type free identifiers in $(\texttt{lambda}(x).e)$.

$$
R_e[\![\texttt{letrec}\ x_1 = e_1; \ldots; x_n = e_n; \texttt{in}\ e]\!]env_r = R_e[\![e]\!]env'_r
$$

where $env'_r = env_r[x_1 \mapsto R_e[\![e_1]\!]env'_r, \ldots, x_n \mapsto R_e[\![e_n]\!]env'_r]$

$$
R_{pr}[\![pr]\!] = R_e[\![pr]\!]nullenv_r
$$

Figure 4.3: Reference Escape Semantic Functions

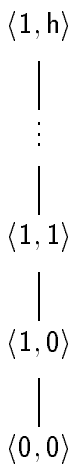

Figure 4.4: The Abstract Basic Reference Escape Domain

4.4 Abstract Reference Escape Semantics

We present a safe and computable abstraction of the exact reference escape semantics defined in the last section that allows an approximation of the exact reference escape behavior for functions to be found at compile-time.

Abstracting Reference Escape Domains

We safely approximate the exact reference escape semantics by abstracting the reference escape semantic subdomains for list type expressions, and by approximating the reference escape semantic functions. For each expression, its corresponding value in the abstract reference escape semantic domain tells us if an interesting reference is definitely not contained in the value of the expression ("non-reference-escape"), or if an interesting reference may be contained in the value of the expression ("possible reference-escape"). The abstract basic reference escape domain is shown in Figure 4.4. The ordering is defined as follows:

$$\langle 0,0\rangle \sqsubseteq \langle 1,0\rangle \sqsubseteq \langle 1,1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,h\rangle$$

The interpretation of the elements of $\hat{B}_r$ is defined as follows:

- $\langle 1, j\rangle$: An interesting reference *may be* contained in the value of the expression, and if $j \geq 1$ then it is a reference to the cons cell at the bottom j^{th} spine of a list. (If $j = 0$ then the object pointed to by the interesting reference is not a cons cell.)

- $\langle 0, 0 \rangle$: *No* interesting reference *is* contained in the value of the expression.

The abstraction of the reference escape semantic subdomains for list type expressions is done by representing lists as finite objects, i.e. by combining the escape pairs of all its elements into a single escape pair. The abstract reference escape semantic domain $\hat{D}_r$ and the domain $\hat{E}_r$ of abstract reference escape environments are defined as follows:

$$\begin{aligned} \hat{D}_r &= \sum_{\tau} \hat{D}_r^{\tau} \quad \text{/* Abstract reference escape semantic domain */} \\ \hat{E}_r &= Id \rightarrow \hat{D}_r \quad \text{/* Domain of abstract reference escape environments */} \end{aligned}$$

The abstract reference escape subdomain $\hat{D}_r^{\tau}$ for expressions of type τ is defined as follows:

$$\begin{aligned} \hat{D}_r^{int} &= \hat{B}_r \times \{err\} && \text{abstract subdomain for integers} \\ \hat{D}_r^{bool} &= \hat{B}_r \times \{err\} && \text{abstract subdomain for booleans} \\ \hat{D}_r^{\tau_1 \rightarrow \tau_2} &= \hat{B}_r \times (\hat{D}_r^{\tau_1} \rightarrow \hat{D}_r^{\tau_2}) && \text{abstract subdomain for functions of type } \tau_1 \rightarrow \tau_2 \\ \hat{D}_r^{\tau\ list} &= \hat{D}_r^{\tau} && \text{abstract subdomain for lists of type } \tau\ list \end{aligned}$$

Abstracting Reference Escape Functions

The abstract reference escape semantic functions are defined as follows:

$$\begin{aligned} \hat{R}_c &: Con \rightarrow \hat{D}_r && \text{/* Abstract reference escape function for constants */} \\ \hat{R}_e &: Exp \rightarrow \hat{E}_r \rightarrow \hat{D}_r && \text{/* Abstract reference escape function for expressions */} \\ \hat{R}_{pr} &: Program \rightarrow \hat{D}_r && \text{/* Abstract reference escape function for programs */} \end{aligned}$$

The abstract reference escape semantic functions are given in Figure 4.5. The abstract value of `cons` returns a single reference escape pair that approximates a list of reference escape pairs by taking the least upper bound of its two arguments. The $\texttt{car}^{\texttt{s}}$ denotes a `car` that is applied to a list with s spines. For each `car` in a program, `s` can be determined statically by type checking. It may be arbitrary large, but is fixed at compile-time. The abstract value of `car` is defined as follows: $\texttt{car}^{\texttt{s}}$ takes a list with `s` spines as an argument, and returns a list with $(\texttt{s}-1)$ spines when $\texttt{s} > 1$ or a non-list object when $\texttt{s} = 1$. In either case, the result cannot contain an interesting reference pointing to the cons cells at the bottom $\texttt{s}^{\texttt{th}}$ spine of a list. The abstract value of `cdr` just returns its argument. $e\hat{n}v_r$ is any abstract reference escape environment in $\hat{E}_r$, and $nul\hat{l}env_r$ is an abstract reference escape environment that maps every identifier to the least element of its abstract reference escape semantic domain.

Safety and Termination

The safety of interpretation under the abstract reference escape semantics can be proved as follows. Let u and v be values of an expression e of type τ in the abstract reference escape

$$
\begin{array}{lcl}
\hat{R}_c[\![c]\!] & = & \langle\langle 0,0\rangle, err\rangle, c = \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\} \\
\hat{R}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \perp_\tau \text{ (The bottom element in } \hat{D}_r^\tau.) \\
\hat{R}_c[\![\texttt{cons}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\
\hat{R}_c[\![\texttt{car}^{\texttt{s}}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\mathrm{cut}^{\texttt{s}}(x)\rangle \\
 & & \text{where } \mathrm{cut}^{\texttt{s}}(z) = \text{if } (z_{(1)(2)} = s) \text{ then } \langle\langle 0,0\rangle, z_{(2)}\rangle \text{ else } z \\
\hat{R}_c[\![\texttt{cdr}]\!] & = & \langle\langle 0,0\rangle, \lambda x.x\rangle \\
\hat{R}_c[\![\texttt{null}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle\langle 0,0\rangle, err\rangle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\hat{R}_e[\![c]\!]e\hat{n}v_r & = & \hat{R}_c[\![c]\!] \\
\hat{R}_e[\![x]\!]e\hat{n}v_r & = & e\hat{n}v_r[\![x]\!] \\
\hat{R}_e[\![e_1 + e_2]\!]e\hat{n}v_r & = & \langle\langle 0,0\rangle, err\rangle \text{ /* same for } e_1 - e_2 \text{ and } e_1 = e_2 \text{ */} \\
\hat{R}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\hat{n}v_r & = & (\hat{R}_e[\![e_2]\!]e\hat{n}v_r) \sqcup (\hat{R}_e[\![e_2]\!]e\hat{n}v_r) \\
\hat{R}_e[\![e_1 e_2]\!]e\hat{n}v_r & = & (\hat{R}_e[\![e_1]\!]e\hat{n}v_r)_{(2)}\ (\hat{R}_e[\![e_2]\!]e\hat{n}v_r) \\
\hat{R}_e[\![\texttt{lambda}(x).e]\!]e\hat{n}v_r & = & \langle\hat{V}, \lambda y.\hat{R}_e[\![e]\!]e\hat{n}v_r[x \mapsto y]\rangle
\end{array}
$$

where

$\hat{V} = \langle 0,0\rangle \sqcup (\bigsqcup_{z \in F} (env_r[\![z]\!])_{(1)})$ and

F = Set of free identifiers in $(\texttt{lambda}(x).e)$.

$$
\hat{R}_e[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n; \texttt{in } e]\!]e\hat{n}v_r = \hat{R}_e[\![e]\!]e\hat{n}v'_r
$$

where $e\hat{n}v'_r = e\hat{n}v_r[x_1 \mapsto \hat{R}_e[\![e_1]\!]e\hat{n}v'_r, \ldots, x_n \mapsto \hat{R}_e[\![e_n]\!]e\hat{n}v'_r]$

$$
\hat{R}_{pr}[\![pr]\!] = \hat{R}_e[\![pr]\!]nul\hat{l}env_r
$$

Figure 4.5: Abstract Reference Escape Semantic Functions

domain $\hat{D}_r$ and the exact reference escape domain D_r, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that the abstract reference escape semantic value u is a *safe* approximation (with respect to reference escape information) for the exact reference escape semantic value v iff

$$(\bigsqcup_{p \text{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k)} p_{(1)}) \sqsubseteq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(1)}$$

for all $k \leq n$ where s_i is a *safe* approximation for t_i for all $i \leq k$.

Theorem 4.1 (Safety) *For any expression e, and environments env_r and $e\hat{n}v_r$ such that for all y, $e\hat{n}v_r[\![y]\!]$ is safe for $env_r[\![y]\!]$, $\hat{R}_e[\![e]\!]e\hat{n}v_r$ is safe (with respect to reference escape information) for $R_e[\![e]\!]env_r$.*

Proof : We can prove by structural induction on expression e.
I. Base Case:

1. $e = c$: $\hat{R}_e[\![c]\!]e\hat{n}v_r = \hat{R}_c[\![c]\!]$ and $R_e[\![c]\!]env_r = R_c[\![c]\!]$. For $c \in \{\ldots,$`-1,0,1,...,` `true, false, nil, null` $\}$, $R_c[\![c]\!] = \hat{R}_c[\![c]\!]$ and thus it clearly holds. For $c =$ `cons`, it holds because if x_1 and x_2 are safe for y_1 and y_2, respectively, then $x_1 \sqcup x_2$ is also safe for a list consisting of y_1 and y_2. It holds for $c =$ `car`$^{\mathrm{s}}$, because if x is safe for y then $\mathrm{cut}^{\mathrm{s}}(x)$ is safe for $\mathrm{first}(y)$. It holds for $c =$ `cdr`, because if x is safe for y then x is clearly safe for $\mathrm{second}(y)$.

2. $e = x$: $\hat{R}_e[\![x]\!]e\hat{n}v_r = e\hat{n}v_r[\![x]\!]$ and $R_e[\![x]\!]env_r = env_r[\![x]\!]$. Since, for all y, $e\hat{n}v_r[\![y]\!]$ is safe for $env_r[\![y]\!]$, it clearly holds.

II. Structural Induction Step: Assume that $\hat{R}_e[\![e]\!]e\hat{n}v_r$ is safe for $R_e[\![e]\!]env_r$ for expressions such as e_0, e_1, e_2, e_3 and e_n. (structural induction hypothesis) Then, we show that $\hat{R}_e[\![e]\!]e\hat{n}v_r$ is safe for $R_e[\![e]\!]env_r$ for $e_1 + e_2$, `if` e_1 `then` e_2 `else` e_3, $e = e_1 e_2$, `lambda`$(x).e_1$ and `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0. This can be proved in the same way as the safety proof of the abstract escape semantics with respect to the exact escape semantics. □

Theorem 4.2 (Termination) *For any (finite) program $pr \in Program$, $\hat{R}_{pr}[\![pr]\!]$ is computable.*

Proof : Every functional in the reference escape domain that is defined by the abstract reference escape semantic functions is composed of the operators such as the least upper bound operator $\sqcup$ and $\mathrm{cut}^{\mathrm{s}}$, which are all monotonic. As before, each subdomain $\hat{D}_r^{\tau}$ is finite. □

4.5 Reference Escapement Testing

Reference escape analysis will determine for a function the relative lifetime of the reference associated with a bound variable with respect to references associated with its occurrences. We perform reference escape analysis on each occurrence of each argument of a function call separately. Thus, at any time we are only interested in whether or not a particular single occurrence of a particular variable escapes. Other objects may escape in the result of a function call, but are ignored by our analysis model. An occurrence is *interesting* if it is the one whose escape behavior we are trying to determine. If a variable has n occurrences, then reference escape analysis will be performed n times, each time treating a different occurrence as interesting.

Consider the example from before:

```
f x y =  letrec g a b = cons a b
         in cons x (g x y)
```

As we discussed previously, each occurrence of x in the body of f denotes the creation of a new reference. To differentiate between the occurrences of x, we label each occurrence differently. In fact, the different occurrences can be considered different parameters to the auxiliary function derived from `f`:

```
f' x1 x2 y =  letrec g a b = cons a b
              in cons x1 (g x2 y)
```

Our abstract reference escape semantics will give the escapement of the parameters to `f'`, and thus of the references corresponding to `x1` and `x2`. We describe below how this analysis proceeds.

Global Reference Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position (i,j) of an interesting reference of a parameter, and an abstract reference escape semantic environment $e\hat{n}v_r$ mapping f to an element of $\hat{D}_r$, the global reference escape test function G_refescape? determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of f could escape f globally. It is defined as follows:

$$\mathsf{G_refescape?}(f,i,j,e\hat{n}v_r) =$$
$$(\hat{R}_e[\![f'\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\hat{n}v_r[f' \mapsto \hat{f}', x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\hat{f}' = \hat{R}_e[\![f']\!]e\hat{n}v_r,$$

$$y_{ij} = \langle\langle 1, s_i\rangle, W^{\tau_i}\rangle, \ /^*\ \text{The } j^{th} \text{ occurrence of } i^{th} \text{ parameter is interesting } ^*/$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0, 0\rangle, W^{\tau_i}\rangle, \ /^*\ \text{Other occurrences of } i^{th} \text{ parameter are not interesting } ^*/$$

and for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0, 0\rangle, W^{\tau_l}\rangle, \ /^*\ \text{Occurrences of other parameters are not interesting } ^*/$$

and τ_i is the type of the i^{th} parameter of f. Only the reference associated with the j^{th} occurrence of the i^{th} parameter is interesting reference, and the references associated with other occurrences of the i^{th} parameter are not interesting. Similarly, the references associated with all occurrences of parameters other than the i^{th} parameter are also not interesting. In order to represent the functional behavior of all possible expressions that could be the i^{th} argument to f, the worst-case behavior is taken. From the result of the global reference escape test function, we can conclude the following:

- If G_refescape?$(f, i, j, e\hat{n}v_r) = 0$ then we conclude that the reference associated with the j^{th} occurrence of the i^{th} argument *does not* escape the function call to f in any possible application of f to n arguments.
- If G_refescape?$(f, i, j, e\hat{n}v_r) = 1$ then it means that the reference associated with the j^{th} occurrence of the i^{th} argument *could* escape the function call in some possible application of f to n arguments.

Local Reference Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in a particular function application $f\ e_1\ \ldots\ e_n$, the position (i, j) of an interesting reference of a parameter, and an abstract reference escape semantic environment $e\hat{n}v_r$ mapping f and free identifiers within e_1 through e_n to elements of $\hat{D}_r$, the global reference escape test function L_refescape? determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of f could escape f globally. It is defined as follows:

$$\mathsf{L_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) =$$
$$(\hat{R}_e[\![f'\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\hat{n}v_r[f' \mapsto \hat{f}', x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\hat{f}' = \hat{R}_e[\![f']\!]e\hat{n}v_r,$$

$$y_{ij} = \langle\langle 1, s_i\rangle, (\hat{R}_e[\![e_i]\!]e\hat{n}v_r)_{(2)}\rangle,$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0, 0\rangle, (\hat{R}_e[\![e_i]\!]e\hat{n}v_r)_{(2)}\rangle,$$

for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0, 0\rangle, (\hat{R}_e[\![e_l]\!]e\hat{n}v_r)_{(2)}\rangle.$$

Then, from the result of the local reference escape test function, we can conclude as follows:

- If $\mathsf{L_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = 0$ then we conclude that the reference associated with the j^{th} occurrence of the i^{th} argument *does not* escape the function call to f in $(f\ e_1 \ldots e_n)$.
- If $\mathsf{L_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = 1$ then it means that the reference associated with the j^{th} occurrence of the i^{th} argument *could* escape the function call $(f\ e_1 \ldots e_n)$.

Examples

As an example, consider a program defined as follows:

```
letrec  map f l = if (null l) then nil
                  else cons (f (car l)) (map f (cdr l));

        sum l = if (null l) then 0
                else (car l) + sum (cdr l);

        addsum x y = cons x (cons y (cons
                     (map (lambda(z). (sum y) + z) x) nil));
in ...
```

We assume that the type of each function is given by

`map` : $(int \rightarrow int) \rightarrow int\ list \rightarrow int\ list$

`sum` : $int\ list \rightarrow int$

`addsum` : $int\ list \rightarrow int\ list \rightarrow (int\ list)list$

From type checking, each `car` in the program can be annotated as follows:

```
map f l = if (null l) then nil
          else cons (f (CAR1 l)) (map f (cdr l));

sum l = if (null l) then 0
        else (CAR1 l) + sum (cdr l);
```

where `CARi` denotes a `car` that takes as its argument a list with `i` spines. The definitions of the reference escape semantic values of `map`, `sum`, and `addsum` are as follows:

$$\begin{array}{lcl} map\ f\ l & = & \langle\langle 0,0\rangle, err\rangle \sqcup (f_{(2)}\ \langle \mathrm{cut}^1(l_{(1)}), l_{(2)}\rangle) \sqcup ((map_{(2)}\ f)_{(2)}\ l) \\ sum\ l & = & \langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \\ addsum\ x\ y & = & x \sqcup y\ \sqcup ((map_{(2)}\ \langle y_{(1)}, \lambda z.\langle\langle 0,0\rangle, err\rangle\rangle)_{(2)}\ x) \end{array}$$

Since *map* is defined recursively, the meaning of *map* is found by a fixpoint iteration:

$$\begin{array}{lcl} map^{(0)}\ f\ l & = & \langle\langle 0,0\rangle, err\rangle \\ map^{(1)}\ f\ l & = & \langle\langle 0,0\rangle, err\rangle \sqcup (f_{(2)}\ \langle \mathrm{cut}^1(l_{(1)}), l_{(2)}\rangle) \sqcup\ ((map^{(0)}_{(2)}\ f)_{(2)}\ l) \\ & = & f_{(2)}\ \langle \mathrm{cut}^1(l_{(1)}), l_{(2)}\rangle \\ map^{(2)}\ f\ l & = & \langle\langle 0,0\rangle, err\rangle \sqcup (f_{(2)}\ \langle \mathrm{cut}^1(l_{(1)}), l_{(2)}\rangle) \sqcup\ ((map^{(1)}_{(2)}\ f)_{(2)}\ l) \\ & = & f_{(2)}\ \langle \mathrm{cut}^1(l_{(1)}), l_{(2)}\rangle \end{array}$$

Since $map^{(1)} = map^{(2)}$, we have that

$$map = \langle\langle 0,0\rangle, \lambda f.\langle f_{(1)}, \lambda l.f_{(2)}\ \langle \mathrm{cut}^1(l_{(1)}), l_{(2)}\rangle\rangle\rangle$$

$$\begin{array}{lcl} addsum\ x\ y & = & x \sqcup y\ \sqcup ((map_{(2)}\ \langle y_{(1)}, \lambda z.\langle\langle 0,0\rangle, err\rangle\rangle)_{(2)}\ x) \\ & = & x \sqcup y\ \sqcup (\lambda z.\langle\langle 0,0\rangle, err\rangle)\ \langle \mathrm{cut}^1(x_{(1)}), x_{(2)}\rangle \\ & = & x \sqcup y\ \sqcup\ \langle\langle 0,0\rangle, err\rangle \\ & = & x \sqcup y \end{array}$$

The auxiliary functions `map'` and `addsum'` for `map` and `addsum` are defined as follows:

```
map' f1 f2 l1 l2 l3 = if (l1=nil) then nil
                      else cons (f1 (car l2))
                                (map f2 (cdr l3));
```

```
addsum' x1 x2 y1 y2 = cons x1 (cons y1
                (cons (map (lambda(z).(sum y2)+z) x2) nil));
```

Note that `map'` is not recursively defined. The definitions of the reference escape semantic values of `map'`, and `addsum'` are given as follows (without a fixpoint iteration):

$$\begin{aligned} map'\ f1\ f2\ l1\ l2\ l3 &= \langle\langle 0,0\rangle, err\rangle \sqcup (f1_{(2)}\ \langle \mathrm{cut}^1(l2_{(1)}), l2_{(2)}\rangle) \sqcup ((map_{(2)}\ f2)_{(2)}\ l3) \\ &= \langle\langle 0,0\rangle, err\rangle \sqcup (f1_{(2)}\ \langle \mathrm{cut}^1(l2_{(1)}), l2_{(2)}\rangle) \\ &\quad \sqcup (f2_{(2)}\ \langle \mathrm{cut}^1(l3_{(1)}), l3_{(2)}\rangle) \\ addsum'\ x1\ x2\ y1\ y2 &= x1 \sqcup y1 \end{aligned}$$

Let $e\hat{n}v_r = [\texttt{map} \mapsto map, \texttt{add} \mapsto add, \texttt{addsum} \mapsto addsum]$. Then,

$$\begin{aligned} \texttt{G_refescape?}(\texttt{addsum}, 1, 1, e\hat{n}v_r) &= (\hat{R}_e[\![\texttt{addsum' x1 x2 y1 y2}]\!]e\hat{n}v'_r)_{(1)(1)} \\ &= 1 \end{aligned}$$

where $e\hat{n}v'_r = e\hat{n}v_r[\texttt{addsum'} \mapsto addsum', \texttt{x1} \mapsto \langle\langle 1,1\rangle, err\rangle, \texttt{x2}, \texttt{y1}, \texttt{y2} \mapsto \langle\langle 0,0\rangle, err\rangle]$. Thus, we can conclude that the reference associated with the first occurrence `x1` of the first parameter `x` of `addsum` escapes. And,

$$\begin{aligned} \texttt{G_refescape?}(\texttt{addsum}, 1, 2, e\hat{n}v_r) &= (\hat{R}_e[\![\texttt{addsum' x1 x2 y1 y2}]\!]e\hat{n}v'_r)_{(1)(1)} \\ &= 0 \end{aligned}$$

where $e\hat{n}v'_r = e\hat{n}v_r[\texttt{addsum'} \mapsto addsum', \texttt{x2} \mapsto \langle\langle 1,1\rangle, err\rangle, \texttt{x1}, \texttt{y1}, \texttt{y2} \mapsto \langle\langle 0,0\rangle, err\rangle]$. Thus, we can conclude that the reference associated with the second occurrence `x2` of the first parameter `x` of `addsum` does not escape.

$$\begin{aligned} \texttt{G_refescape?}(\texttt{addsum}, 2, 1, e\hat{n}v_r) &= (\hat{R}_e[\![\texttt{addsum' x1 x2 y1 y2}]\!]e\hat{n}v'_r)_{(1)(1)} \\ &= 1 \end{aligned}$$

where $e\hat{n}v'_r = e\hat{n}v_r[\texttt{addsum'} \mapsto addsum', \texttt{y1} \mapsto \langle\langle 1,1\rangle, err\rangle, \texttt{x1}, \texttt{x2}, \texttt{y2} \mapsto \langle\langle 0,0\rangle, err\rangle]$.

$$\begin{aligned} \texttt{G_refescape?}(\texttt{addsum}, 2, 2, e\hat{n}v_r) &= (\hat{R}_e[\![\texttt{addsum' x1 x2 y1 y2}]\!]e\hat{n}v'_r)_{(1)(1)} \\ &= 0 \end{aligned}$$

where $e\hat{n}v'_r = e\hat{n}v_r[\texttt{addsum'} \mapsto addsum', \texttt{y2} \mapsto \langle\langle 1,1\rangle, err\rangle, \texttt{x1}, \texttt{x2}, \texttt{y1}, \mapsto \langle\langle 0,0\rangle, err\rangle]$. Thus, similarly, we can conclude that the reference associated with the first occurrence `y1` of the second parameter `y` of `addsum` escapes, but the reference associated with the second occurrence `y2` of the second parameter `y` of `addsum` does not escape.

$$\begin{aligned}\mathsf{G_refescape?}(\mathtt{map},1,1,\hat{en}v_r) &= (\hat{R}_e[\![\mathtt{map'\ f1\ f2\ l1\ l2\ l3}]\!]\hat{en}v'_r)_{(1)(1)}\\ &= 0\end{aligned}$$

where

$$\begin{aligned}\hat{en}v'_r = \hat{en}v_r[&\mathtt{map'} \mapsto map', \mathtt{f1} \mapsto \langle\langle 1,0\rangle, \lambda z.\langle z_{(1)}, err\rangle\rangle,\\ &\mathtt{f2} \mapsto \langle\langle 0,0\rangle, \lambda z.\langle z_{(1)}, err\rangle\rangle, \mathtt{l1},\mathtt{l2},\mathtt{l3} \mapsto \langle\langle 0,0\rangle, err\rangle].\end{aligned}$$

and

$$\begin{aligned}\mathsf{G_refescape?}(\mathtt{map},1,2,\hat{en}v_r) &= (\hat{R}_e[\![\mathtt{map'\ f1\ f2\ l1\ l2\ l3}]\!]\hat{en}v'_r)_{(1)(1)}\\ &= 0\end{aligned}$$

where

$$\begin{aligned}\hat{en}v'_r = \hat{en}v_r[&\mathtt{map'} \mapsto map', \mathtt{f1} \mapsto \langle\langle 0,0\rangle, \lambda z.\langle z_{(1)}, err\rangle\rangle,\\ &\mathtt{f2} \mapsto \langle\langle 1,0\rangle, \lambda z.\langle z_{(1)}, err\rangle\rangle, \mathtt{l1},\mathtt{l2},\mathtt{l3} \mapsto \langle\langle 0,0\rangle, err\rangle].\end{aligned}$$

In the same way, we also can conclude that the references associated with the occurrences `f1, f2` of the first parameter `f` of `map` do not escape.

4.6 Improving Precision of Reference Escapement

We present a method of improving the precision of reference escape information that is obtainable through the reference escape analysis by using position information about interesting references in a list structure. We describe another safe and computable abstraction of the exact reference escape semantics that gives more precise information about reference escapement. The basic improved abstract reference escape domain is constructed by extending the basic reference escape domain to include additional information about both spine level and position. We safely approximate the exact reference escape semantics by abstracting the reference escape semantic subdomains for list type expressions, and by approximating reference escape semantic functions. Abstraction of the reference escape semantic subdomains for list type expressions is done by representing lists as finite objects, i.e. by combining the reference escape pairs of all its elements into a single reference escape pair.

For each expression, its corresponding value in the improved abstract reference escape domain tells us if an interesting reference is definitely not contained in the value of the expression ("non-reference-escape"), or if an interesting reference may be contained at some position in the value of the expression ("possible reference-escape and where"). The basic

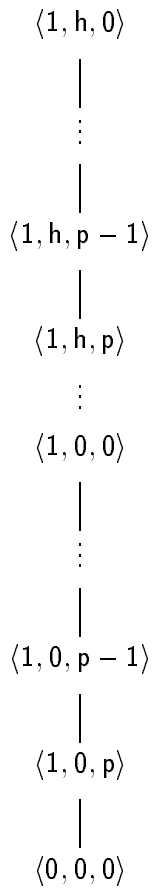

Figure 4.6: The Improved Abstract Basic Reference Escape Domain

improved abstract reference escape domain, $\tilde{B}_r$, for some fixed p is shown in Figure 4.6. The ordering is defined as follows:

$$\langle 0,0,0\rangle \sqsubseteq \langle 1,0,p\rangle \sqsubseteq \langle 1,0,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,0,0\rangle \ldots \langle 1,h,p\rangle \sqsubseteq \langle 1,h,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,h,0\rangle$$

The interpretation of the elements of $\tilde{B}_r$ is defined as follows:

- $\langle 1,j,k\rangle$: An interesting reference *may be* contained in the value of the expression, and if $j \geq 1$ then it is a reference to a cons cell at the bottom j^{th} spine of a list and it *may* occur only at $\geq k$ position. (If $j = 0$ then the object pointed by the interesting reference is not a cons cell.)
- $\langle 0,0,0\rangle$: *No* interesting reference *is* contained in the value of the expression.

The improved abstract reference escape semantic domain $\tilde{D}_r$ and the domain $\tilde{E}_r$ of improved abstract reference escape environments are defined as follows:

$$\begin{array}{lcll} \tilde{D}_r & = & \sum_\tau \tilde{D}_r^\tau & \text{/* Improved abstract reference escape semantic domain */} \\ \tilde{E}_r & = & Id \rightarrow \tilde{D}_r & \text{/* Domain of improved reference escape environments */} \end{array}$$

The improved abstract reference escape subdomain $\tilde{D}_r^\tau$ for expressions of type τ is defined as follows:

$$\begin{array}{lcll} \tilde{D}_r^{int} & = & \tilde{B}_r \times \{err\} & \text{improved abstract subdomain for integers} \\ \tilde{D}_r^{bool} & = & \tilde{B}_r \times \{err\} & \text{improved abstract subdomain for booleans} \\ \tilde{D}_r^{\tau_1 \rightarrow \tau_2} & = & \tilde{B}_r \times (\tilde{D}_r^{\tau_1} \rightarrow \tilde{D}_r^{\tau_2}) & \text{improved abstract subdomain for functions} \\ \tilde{D}_r^{\tau\ list} & = & \tilde{D}_r^\tau & \text{improved abstract subdomain for lists} \end{array}$$

The abstract reference escape semantic functions are defined as follows:

$$\begin{array}{lcll} \tilde{R}_c & : & Con \rightarrow \tilde{D}_r & \text{/* Improved semantic function for constants */} \\ \tilde{R}_e & : & Exp \rightarrow \tilde{E}_r \rightarrow \tilde{D}_r & \text{/* Improved semantic function for expressions */} \\ \tilde{R}_{pr} & : & Program \rightarrow \tilde{D}_r & \text{/* Improved semantic function for programs */} \end{array}$$

The improved abstract reference escape semantic function $\tilde{R}_c : Con \rightarrow \tilde{D}_r$ that gives reference escape meaning to constants is given in Figure 4.7. The improved abstract value of `cons` takes two arguments, and returns an improved reference escape pair of their least upper bound by updating the position information of any interesting reference in the resulting list. The improved abstract value of $\texttt{car}^{\texttt{s}}$ returns its argument according to the position of any interesting object. $\texttt{car}^{\texttt{s}}$ takes a list with `s` spines and returns the element that was in the 0^{th} position and has $(\texttt{s}-1)$ spines. Thus, the result cannot contain an interesting reference

$$
\begin{array}{lcl}
\tilde{R}_c[\![c]\!] & = & \langle\langle 0,0,0\rangle, err\rangle,\ \ c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\} \\
\tilde{R}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (The bottom element in } \tilde{D}^\tau_r.) \\
\\
\tilde{R}_c[\![\texttt{cons}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\text{ipush}(x,y)\rangle\rangle \\
 & & \text{where} \\
 & & \quad \text{ipush}(x,y) = \text{ if } (x_{(1)(1)} = 1) \text{ then} \\
 & & \qquad\qquad \langle\langle 1, x_{(1)(2)} \sqcup y_{(1)(2)}, 0\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
 & & \qquad\quad \text{elseif } (y_{(1)(1)} = 1) \text{ then} \\
 & & \qquad\qquad \langle\langle 1, y_{(1)(2)}, \min(y_{(1)(3)} + 1, p)\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
 & & \qquad\quad \text{else } \langle\langle 0,0,0\rangle, x_{(2)} \sqcup y_{(2)}\rangle \\
\tilde{R}_c[\![\texttt{car}^{\texttt{s}}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\text{icut}^{\texttt{s}}(x)\rangle \\
 & & \text{where} \\
 & & \quad \text{icut}^{\texttt{s}}(z) = \text{ if } (s \leq z_{(1)(2)}) \text{ or } (z_{(1)(3)} > 0) \text{ then } \langle\langle 0,0,0\rangle, z_{(2)}\rangle \\
 & & \qquad\qquad \text{else } z \\
\tilde{R}_c[\![\texttt{cdr}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\text{rrest}(x)\rangle \\
 & & \text{where} \\
 & & \quad \text{rrest}(z) = \langle\langle z_{(1)(1)}, z_{(1)(2)}, \max[z_{(1)(3)} - 1, 0]\rangle, z_{(2)}\rangle \\
\tilde{R}_c[\![\texttt{null}]\!] & = & \langle\langle 0,0,0\rangle, \lambda x.\langle\langle 0,0,0\rangle, err\rangle\rangle
\end{array}
$$

Figure 4.7: Improved Abstract Reference Escape Semantic Function

pointing to a cons cell at the bottom $\mathtt{s}^{th}$ spine of a list or that occurred at a position ≥ 1 in the original list. The improved abstract value of `cdr` also updates the position information appropriately. Note that the improved abstract reference escape semantic function $\tilde{R}_c$ for `cons`, `car` and `cdr` provides more precise escape information than the abstract reference escape semantic function $\hat{R}_c$.

The improved abstract reference escape semantic functions $\tilde{R}_e$ and $\tilde{R}_{pr}$ are defined identically to $\hat{R}_e$ and $\hat{R}_{pr}$, respectively - except for their value in the basic domain.

Let u and v be values of an expression e of type τ in the improved abstract reference escape domain $\tilde{D}_r$ and the exact reference escape domain D_r, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that the improved abstract reference escape semantic value u is a *safe* approximation of the exact reference escape semantic value v iff

$$(\bigsqcup_{p \texttt{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k)} p_{(1)}) \sqsubseteq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(1)}$$

and

$$(MIN_{p \texttt{ in } \mathrm{NAP}_k(v,t_1,\ldots,t_k) \& p_{(1)}=1} \text{Position of } p) \geq (\mathrm{NAP}_k(u, s_1, \ldots, s_k))_{(1)(3)}$$

for all $k \leq n$ where s_i is a *safe* approximation for t_i for all $i \leq k$.

Theorem 4.3 (Safety) *For any expression e, and environments env_r and $e\tilde{n}v_r$ such that for all y, $e\tilde{n}v_r[\![y]\!]$ is safe for $env_r[\![y]\!]$, $\tilde{R}_e[\![e]\!]e\tilde{n}v_r$ is safe (with respect to reference escape information) for $R_e[\![e]\!]env_r$.*

Proof : We can prove by structural induction on expression e.

I. Base Case:

1. $e = c$: $\tilde{R}_e[\![c]\!]e\tilde{n}v_r = \tilde{R}_c[\![c]\!]$ and $R_e[\![c]\!]env_r = R_c[\![c]\!]$. For $c \in$ { `...,-1,0,1,..., true, false, nil, null` } , $R_c[\![c]\!] = \tilde{R}_c[\![c]\!]$ and thus it clearly holds. For $c =$ `cons`, it holds because if x_1 and x_2 are safe for y_1 and y_2, respectively, then $\mathrm{ipush}(x_{(1)}, x_2)$ is also safe for a list consisting of y_1 and y_2. It holds for $c = \mathtt{car}^{\mathtt{s}}$, because if x is safe for y then $\mathrm{icut}^{\mathrm{s}}(x)$ is safe for $\mathrm{first}(y)$. It holds for $c = \mathtt{cdr}^{\mathtt{s}}$, because if x is safe for y then $\mathrm{rrest}(x)$ is safe for $\mathrm{second}(y)$.

2. $e = x$: $\tilde{R}_e[\![x]\!]e\tilde{n}v_r = e\tilde{n}v_r[\![x]\!]$ and $R_e[\![x]\!]env_r = env_r[\![x]\!]$. Since, for all y, $e\tilde{n}v_r[\![y]\!]$ is safe for $env_r[\![y]\!]$, it clearly holds.

II. Structural Induction Step: Assume that $\tilde{R}_e[\![e]\!]e\tilde{n}v_r$ is safe for $R_e[\![e]\!]env_r$ for expressions such as e_0, e_1, e_2, e_3 and e_n. (structural induction hypothesis) Then, we show that

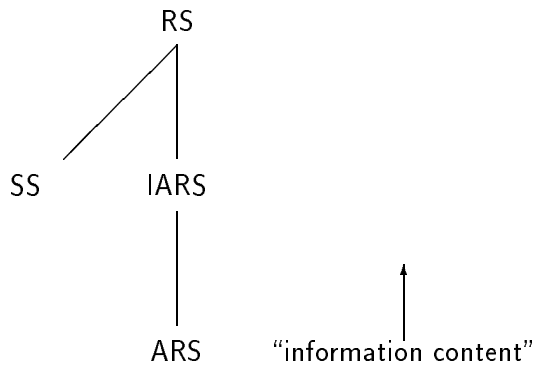

Figure 4.8: Relationship among Reference Escape Semantics

$\tilde{R}_e[\![e]\!]e\tilde{n}v_r$ is safe for $R_e[\![e]\!]envr$ for $e_1 + e_2$, `if` e_1 `then` e_2 `else` e_3, $e = e_1 e_2$, `lambda`$(x).e_1$ and `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0. This can be proved in an exactly same way as the proof of safety of the abstract reference escape semantics. □

Theorem 4.4 (Termination) *For any (finite) program* $pr \in Program$, $\tilde{R}_{pr}[\![pr]\!]$ *is computable, i.e. always terminates in finite number of steps.*

Proof : Every functional in the improved reference escape domain that is defined via the abstract improved extended escape semantic functions is composed of the operators such as the least upper bound operator $\sqcup$, ipush, icut^s, and rrest. ipush^s, icut^s and rrest are monotonic operators. Since the composition of monotonic functions is also monotonic, every functional is monotonic. □

Theorem 4.5 (Precision Improvement) *The abstract reference escape semantics is equivalent (with respect to their reference escapement information content) to the improved abstract reference escape semantics with* $p = 0$. *Thus, the improved abstract reference escape semantics with some* $p > 0$ *provides more precise escape information than the abstract reference escape semantics.*

Proof : This can be proved in the similar way to the proof of Theorem 2.5. □

The relationship among the standard semantics (SS), the non-standard exact reference escape semantics (RS), the abstract reference escape semantics (ARS) and the improved abstract reference escape semantics (IARS) is shown in Figure 4.8.

Improved Global Reference Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position (i,j) of an interesting reference of a parameter, and an improved abstract reference escape semantic environment $e\tilde{n}v_r$ mapping f to an element of $\tilde{D}_r$, the improved global reference escape test function G_irefescape? which determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of f could escape f globally is defined as follows:

$$\mathsf{G_irefescape?}(f,i,j,e\tilde{n}v_r) =$$
$$(\tilde{R}_e[\![f'\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\tilde{n}v_r[f' \mapsto \tilde{f}', x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\tilde{f}' = \tilde{R}_e[\![f']\!]e\tilde{n}v_r,$$

$$y_{ij} = \langle\langle 1, s_i, 0\rangle, W^{\tau_i}\rangle,\ \text{/* The } j^{th} \text{ occurrences of } i^{th} \text{ parameter is interesting */}$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0, 0, 0\rangle, W^{\tau_i}\rangle,\ \text{/* Other occurrences of } i^{th} \text{ parameter are not interesting */}$$

for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0, 0, 0\rangle, W^{\tau_l}\rangle,\ \text{/* Occurrences of other parameters are not interesting */}$$

and τ_i is the type of the i^{th} parameter of f. The result of the global improved reference escape test function is interpreted as follows:

- If $\mathsf{G_irefescape?}(f,i,j,e\tilde{n}v_r) = 0$ then we can conclude that the reference associated with the j^{th} occurrence of the i^{th} argument *does not* escape the function call to f in any possible application of f to n arguments.
- If $\mathsf{G_irefescape?}(f,i,j,e\tilde{n}v_r) = 1$ then the reference associated with the j^{th} occurrence of the i^{th} argument *could* escape the function call to f in some possible application of f to n arguments.

Improved Local Reference Escape Test

Given a function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in a particular function application $f\ e_1\ \ldots\ e_n$, the position (i,j) of an interesting reference of a parameter, and an improved abstract reference escape semantic environment $e\tilde{n}v_r$ mapping f and the free identifiers

within e_1 through e_n to elements of $\tilde{D}_r$, the improved global reference escape test function L_irefescape? which determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of f could escape f locally is defined as follows:

$$\mathsf{L_irefescape?}(f, i, j, e_1, \ldots, e_n, e\tilde{n}v_r) =$$
$$(\tilde{R}_e[\![f'\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\tilde{n}v_r[f' \mapsto \tilde{f}', x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\tilde{f}' = \tilde{R}_e[\![f']\!]e\tilde{n}v_r,$$

$$y_{ij} = \langle\langle 1, s_i, 0\rangle, (\tilde{R}_e[\![e_i]\!]e\tilde{n}v_r)_{(2)}\rangle,$$

s_i is the number of spines of the i^{th} parameter of f (if it is a list type, otherwise s_i is 0), for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0, 0, 0\rangle, (\tilde{R}_e[\![e_i]\!]e\tilde{n}v_r)_{(2)}\rangle,$$

for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0, 0, 0\rangle, (\tilde{R}_e[\![e_l]\!]e\tilde{n}v_r)_{(2)}\rangle.$$

The result of the local improved reference escape test function is interpreted as follows:

- If $\mathsf{L_irefescape?}(f, i, j, e_1, \ldots, e_n, e\tilde{n}v_r) = 0$ then we can conclude that the reference associated with the j^{th} occurrence of the i^{th} argument *does not* escape f locally in $(f\ e_1 \ldots\ e_n)$.
- If $\mathsf{L_irefescape?}(f, i, j, e_1, \ldots, e_n, e\tilde{n}v_r) = 1$ then it means that the reference associated with the j^{th} occurrence of the i^{th} argument *could* escape f locally in $(f\ e_1 \ldots\ e_n)$.

4.7 Complexity of Reference Escape Analysis

The abstract interpretation framework for the reference escape analysis that deals with higher-order functional languages with non-flat domains is very similar to the framework for strictness analysis for higher-order functional languages with non-flat domains. Both analyses use a k-element domain where k is fixed and greater than 2 as their basic abstract domains, respectively. Like any other analysis based on abstract interpretation, the major complexity of our analysis comes from finding the fixpoints of recursive functions in the abstract semantic domains. In our analysis, the reference escape testing is performed on

each reference associated with each occurrence of a parameter of a function separately using its auxiliary function. However, since the auxiliary function for a function is never recursive even if the original function is a recursive function, the process of finding a fixpoint is needed only for an original function, but is never needed for its auxiliary function. Thus, the order of time complexity of reference escape analysis is the same as that of strictness analysis for higher-order languages with non-flat domains, which is exponential.

Chapter 5

Order-of-Demand Analysis

In lazy evaluation, arguments to a function are not evaluated unless and until their values are demanded, and are evaluated only once upon the first demand. Their values are then saved to be used for subsequent demands, thus avoiding reevaluation. Exact information about the strictness of arguments, the order of evaluation among arguments, and the evaluation status of arguments when demanded is generally unknown at compile-time. If inferred at compile-time, such information can be useful for a number of optimizations for lazy evaluation.

In this chapter, we present a method for statically inferring a range of information including strictness, evaluation-order, and evaluation-status information in a higher-order, monomorphic, non-strict functional language with lazy evaluation. This method is based on a compile-time analysis called *order-of-demand analysis* which provides safe information about the order in which the values of bound variables are demanded. First, we introduce a non-standard denotational semantics called *before semantics* that describes the actual order-of-demand behavior, but is incomputable at compile-time. A method of approximating the exact before semantics which is safe with respect to the exact before semantics and is computable at compile-time is then presented. Based on this *abstract before semantics*, we describe algorithms which determine order-of-demand information and the complexity of the order-of-demand analysis. Finally, extensions of the order-of-demand analysis to a parallel lazy evaluation model, to an optimized lazy evaluation model using strictness information, and to non-flat domains are discussed.

5.1 Order of Demand under Lazy Evaluation

We present a method for detecting at compile-time a range of information including strictness, evaluation-order, and evaluation-status information in a higher-order functional language being implemented using lazy evaluation. The range of information can be reformulated in terms of information on order of demand as follows:

- *Evaluation Status*: Given an occurrence x_i of a variable x in the body of a function f, for each possible execution of the body of f
  - If there exists another occurrence x_j of x such that x_j is demanded before x_i (may be several x_j's in different paths), then we know that x must have been evaluated by the time x_i is encountered.
  - If there exists no occurrence x_j of x such that x_j is demanded before x_i, then we know that x must have not been evaluated by the time x_i is encountered.
- *Parameter Evaluation Order*: Given two parameters x and y of a function f, if for every occurrence y_i of y in the body of f there exists an occurrence x_j of x that is demanded before y_i, then we can conclude that x will always be evaluated before y.
- *Strictness*: Given a parameter x of a function f, and an imaginary variable \$ which is demanded just after the call to f is evaluated, if for each possible (terminating) execution of the body of f some occurrence x_i of x was demanded before \$, then f is strict with respect to x.

Consider, for example, the following function:

```
f w x y z = if w=0 then z+x elseif y=0 then z+x
            else (z+y) + f (w-1) x (y-1) z
```

We assume that the primitive functions such as `+, -, =` evaluate their arguments in left-to-right order. For convenience, we represent each occurrence of a `f`'s bound variable as a distinct variable as follows:

```
= if w1=0 then z1+x1 elseif y1=0 then z2+x2
  else (z3+y2) + f (w2-1) x3 (y3-1) z4
```

Our order-of-demand analysis would allow us to determine the following properties of the function `f` at compile time:

- *Evaluation Status*: When the value of `y` is demanded via `y2` and `y3`, respectively, `y` has already been evaluated. But, when it is demanded via `y1`, `y` has not yet been evaluated at the point.

- *Parameter Evaluation Order*: The parameters to `f` are evaluated in the order `w`, `z` and `x`.

- *Strictness*: `f` is strict in `w`, `x` and `z`, but is not strict in `y`.

The method is based on a compile-time analysis called *order-of-demand analysis* which provides safe information about the order in which the values of variables are demanded. Rather than try to determine the relative order-of-demand between all possible pairs of occurrences, it turns out to be useful and more efficient to define an order-of-demand relation between *sets* of occurrences. We define a relation between two sets of occurrences of parameters of a function which specifies the order-of-demand property between them.

Definition 5.1 Given two *non-empty disjoint sets* X and Y of labeled occurrences of variables of a function and an expression e, we define an order-of-demand relation between X and Y during evaluation of an expression e (which contain the occurrences) as follows:

- $Y \prec X$, pronounced Y *before* X, if for each occurrence $x \in X$, *either* there exists an occurrence $y \in Y$ that is demanded before x *or* x is not demanded during the evaluation of e.

- If $Y \prec X$ holds for *any* possible application of f to n arguments then we say that $Y \prec X$ *globally*. If $Y \prec X$ holds for a function application of f to some particular n arguments then we say that $Y \prec X$ locally.

For example, some properties of $\prec$ among occurrences of variables in `f` given above are:

- { `y1,y3` } $\prec$ { `y2` }, { `y1,y2` } $\prec$ { `y3` }, and { `y2,y3` } $\not\prec$ { `y1` }.

- { `w1,w2` } $\prec$ { `z1,z2,z3,z4` } and { `z1,z2,z3,z4` } $\prec$ { `x1,x2,x3` }.

- { `w1,w2` } $\prec$ { \$ }, { `x1,x2,x3` } $\prec$ { \$ }, { `z1,z2,z3,z4` } $\prec$ { \$ }, and { `y1,y2,y3` } $\not\prec$ { \$ } where \$ denotes an occurrence of an imaginary variable that is demanded after `(f w x y z)` is evaluated.

The order-of-demand analysis described here answers the following question: Given a function and two sets X and Y of occurrences of the function's bound variables, which of the following relations holds, if any : $Y \prec X$ or $X \prec Y$. We will develop a method

for a higher-order, monomorphic, non-strict functional language to answer the following questions at compile-time:

- Given a function, what is the order of demand between the parameters or locally defined objects within the function, globally?
- Given a function in a particular application, what is the order of demand between the parameters or locally defined objects locally?

Function Transformation

The first step in the order-of-demand analysis is to differentiate between the (textual) occurrences of each bound variable in a function definition. Although this seems strange at first, it quickly becomes apparent that each occurrence of a variable denotes a different instance in which the value of the variable could be demanded. Roughly speaking, the order-of-demand analysis that we describe can determine the relative order in which the values of various occurrences of a variable, or several variables, are demanded. In order to make each occurrence of a bound variable distinct, we introduce an auxiliary function f' for each function $f\ x_1 \ldots x_n = body_f$ based on the same transformation described in Chapter 4. For example, the auxiliary function `f'` of `f` given above is defined as follows:

```
f' w1 w2 x1 x2 x3 y1 y2 y3 z1 z2 z3 z4
          = if w1=0 then z1+x1 elseif y1=0 then z2+x2
            else (z3+y2) + f (w2-1) x3 (y3-1) z4
```

5.2 Before Analysis

In this section, we present an order-of-demand analysis called *before analysis* which provides safe compile-time information about before demand based on an abstract interpretation technique. We assume that the underlying evaluation model for the higher-order non-strict functional language is sequential lazy evaluation. That is, the evaluation of arguments of strict primitive functions are predefined in a sequential order (either left-to-right or right-to-left) at compile-time and no optimization using strictness information is applied. The cases in which the underlying evaluation model is a parallel lazy evaluation model where arguments of strict primitive functions may be evaluated in parallel, and an optimized lazy evaluation using strictness information in which all strict arguments of a function are evaluated at the time of a call to the function either sequentially or in parallel will be discussed in section 5.5.

5.2.1 Exact Before Semantics

We introduce an exact, but incomputable, *non-standard* denotational semantics called *before semantics*, which exactly describes the actual $\prec$ relation between any two occurrences of a parameter of a function in a program. Since the exact before relation during a program's execution depends on the standard values themselves (e.g. the standard value of the predicate part of a conditional will determines which alternative will be taken), any exact before semantics needs to contain the standard meanings of expressions as well as before information.

Two sets of occurrences of parameters of a function will be analyzed separately to determine the function's before behavior. We say that two particular sets of occurrences of parameters of a function *interesting* sets if we are trying to determine the before relation between them. Thus, our before semantics is defined in terms of two interesting sets X and Y of occurrences of variables inside a function. The exact before semantics determines, given an expression e and two interesting sets X and Y of occurrences of variables that are contained in e, whether the relation $X \prec Y$ holds as a result of the evaluation of e.

Representing Before Information

For each expression, its corresponding value in the before semantic domain should indicate whether the relation Y before X holds during evaluation or not. Under the non-standard before semantics, we represent the meaning of an expression e, with respect to interesting sets X and Y, as a pair in the non-standard before semantic domain D_t, called a before pair,

1. whose first element denotes the before information during evaluation of the expression, and

2. whose second element denotes the functional behavior of the expression e when it is applied to another expression.

The first component of the pair is an element of a domain called a *basic before domain*, B_t, which is a three-element domain of 0, 1 and 2 ordered by $0 \sqsubseteq 1 \sqsubseteq 2$ as shown in Figure 5.1. The elements in B_t are interpreted as follows:

- 2 : For each occurrence $x \in X$, x is demanded and no occurrence $y \in Y$ is demanded before x during the evaluation of e.

Figure 5.1: The Basic Before Domain

- 1: For each occurrence $x \in X$, neither x nor $y \in Y$ is demanded during the evaluation of e.
- 0 : For each occurrence $x \in X$, *either* x is demanded and an occurrence $y \in Y$ is demanded before x *or* x is not demanded and an occurrence $y \in Y$ is demanded during the evaluation of e.

The second component of the pair is a function over D_t, whose meaning is the functional behavior of the expression e when it is applied to another expression. For expressions which have no higher-order behavior, err, a function that can never be applied, is used.

Before Semantic Domains

The before semantic domain D_t and the before environment E_t are defined as follows:

$$\begin{aligned} D_t &= \sum_{\tau} D_t^{\tau} \qquad /^* \text{ Before semantic domain } ^*/ \\ E_t &= Id \rightarrow D_t \quad /^* \text{ Domain of before environments } ^*/ \end{aligned}$$

The before domain D_t is a sum domain consisting of each subdomain for each type. The before subdomain D_t^{τ} for expressions of type τ is defined as follows:

$$\begin{aligned} D_t^{int} &= B_t \times \{err\} && \text{subdomain for integers} \\ D_t^{bool} &= B_t \times \{err\} && \text{subdomain for booleans} \\ D_t^{\tau_1 \rightarrow \tau_2} &= B_t \times (D_t^{\tau_1} \rightarrow D_t^{\tau_2}) && \text{subdomain for functions of type } \tau_1 \rightarrow \tau_2 \end{aligned}$$

Before Semantic Functions

We introduce the non-standard before semantic functions, T_c, T_e, and T_{pr}, which give non-standard before meaning to constants, expressions and programs, respectively.

$$
\begin{array}{lcll}
T_c & : & Con \rightarrow D_t & /* \text{ Before semantic function for constants } */ \\
T_e & : & Exp \rightarrow E_t \rightarrow D_t & /* \text{ Before semantic function for expressions } */ \\
T_{pr} & : & Program \rightarrow D_t & /* \text{ Before semantic function for programs } */
\end{array}
$$

We define a binary operator $\triangleright$ which reflects the notion of sequential evaluation of two expressions as follows:

$$
\begin{array}{l}
\triangleright : B_t \rightarrow B_t \rightarrow B_t \\
U \triangleright V \stackrel{\text{def}}{=} \text{ if } (U = 1) \text{ then } V \text{ else } U
\end{array}
$$

Consider an expression e which consists of two subexpressions e_1 and e_2 such that the evaluation of e_1 is followed by the evaluation of e_2. If the value of e_1 in B_t with respect to X and Y is 0 then, during the evaluation of e_1, any demand to an occurrence $x \in X$ is preceded by a demand to an occurrence $y \in Y$ and, furthermore, at least one occurrence in Y is demanded. Thus, regardless of the value of e_2 in B_t, the *before* information of e is always 0.

If the value of e_1 in B_t with respect to X and Y is 2, then, during the evaluation of e_1, any demand to an occurrence $y \in Y$ is preceded by a demand to an occurrence $x \in X$ and, furthermore, at least one occurrence in X is demanded. Thus, regardless of the value of e_2 in B_t, the *before* information of e is always 2.

If the value of e_1 in B_t with respect to X and Y is 1, then no occurrence $o \in X \cup Y$ is demanded during evaluation of e_1. Therefore, the value of e in B_t depends only on, and is equal to, the value of e_2 in B_t. Thus the $\triangleright$ operator reflects precisely the notion of sequential evaluation of two expressions.

The semantic equations for the before semantic functions are expressed in Figure 5.2. **Oracle** is used to resolve the exact behavior of the conditional primitive function, determining which branch would be evaluated at run-time. Since this must rely on the standard value of a predicate, the exact before semantics is not computable at compile-time. env_t is any exact before environment in E_t, and $nullenv_t$ is a before environment that maps every identifier to the least element of its before semantic domain.

5.2.2 Abstract Before Semantics

Since the information that the before semantics provides is uncomputable at compile time, it is not suitable as a basis for compile-time analysis. For use in a compiler, we need a suitable before semantics that will guarantee termination and yet still provide a useful and safe approximation to the exact before semantics. In this section, we present an abstract

$$T_c[\![c]\!] \;=\; \langle 1, err\rangle, c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\}$$

$$T_e[\![c]\!]env_t \;=\; T_c[\![c]\!]$$

$$T_e[\![x]\!]env_t \;=\; env_t[\![x]\!]$$

$$\begin{array}{lcll} T_e[\![e_1 + e_2]\!]env_t & = & \text{let} & l = T_e[\![e_1]\!]env_t \qquad /^* \text{ same for } e_1 - e_2 \text{ and } e_1 = e_2 \; ^*/ \\ & & & r = T_e[\![e_2]\!]env_t \\ & & \text{in} & \langle l_{(1)} \rhd r_{(1)}, err\rangle \end{array}$$

$$\begin{array}{lcll} T_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]env_t & = & \text{let} & p = T_e[\![e_1]\!]env_t \\ & & & c = T_e[\![e_2]\!]env_t \\ & & & a = T_e[\![e_3]\!]env_t \\ & & \text{in} & \text{if } \mathbf{Oracle}(e_1) \text{ then } \langle p_{(1)} \rhd c_{(1)}, c_{(2)}\rangle \\ & & & \text{else } \langle p_{(1)} \rhd a_{(1)}, a_{(2)}\rangle \end{array}$$

$$\begin{array}{lcll} T_e[\![e_1 e_2]\!]env_t & = & \text{let} & f = T_e[\![e_1]\!]env_t \\ & & & ap = f_{(2)}(T_e[\![e_2]\!]env_t) \\ & & \text{in} & \langle f_{(1)} \rhd ap_{(1)}, ap_{(2)}\rangle \end{array}$$

$$T_e[\![\texttt{lambda}(x).e]\!]env_t \;=\; \langle 1, \lambda y.T_e[\![e]\!]env_t[x \mapsto y]\rangle$$

$$T_e[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n; \texttt{in } e]\!]env_t \;=\; T_e[\![e]\!]env'_t$$

where $env'_t = env_t[x_1 \mapsto T_e[\![e_1]\!]env'_t, \ldots, x_n \mapsto T_e[\![e_n]\!]env'_t]$

$$T_{pr}[\![pr]\!] \;=\; T_e[\![pr]\!]nullenv_t$$

Figure 5.2: Before Semantic Functions

Figure 5.3: The Abstract Basic Before Domain

interpretation of the exact before semantics that allows a safe approximation of the exact before behavior to be found at compile-time.

Abstract Before Semantic Domains

For each expression, its corresponding value in the abstract before semantic domain indicates whether the relation of Y before X holds definitely during the evaluation of the expression ("before"), and whether the relation of Y before X may hold during the evaluation of the expression ("maybe before"). As an abstraction of B_t, we define an *abstract basic before domain* $\hat{B}_t$ as a three-element domain of 0, 1 and 2 ordered by $0 \sqsubseteq 1 \sqsubseteq 2$ that is similar to the basic before domain B_t as shown in Figure 5.3. But, the interpretation of elements in $\hat{B}_t$ is defined differently from that of B_t as follows:

- **2** : For each occurrence $x \in X$, *either* x is demanded and an occurrence $y \in Y$ *might* (or might not) be demanded before x, *or* x is not demanded and an occurrence $y \in Y$ *might* (or might not) be demanded during evaluation of e. (In other words, this value denotes the absence of knowledge about the relative order of demands.)

- **1** : For each occurrence $x \in X$, *either* x is demanded and an occurrence $y \in Y$ is demanded before x, *or* x is not demanded and an occurrence $y \in Y$ *might* (or might not) be demanded during evaluation of e.

- **0** : For each occurrence $x \in X$, *either* x is demanded and an occurrence $y \in Y$ is demanded before x, *or* x is not demanded and an occurrence $y \in Y$ is demanded during the evaluation of e.

Note that the value in B_t of an expression e consisting only of the variable x is 2 if $x \in X$, 0 if $x \in Y$, 1 if $x \notin (X \cup Y)$. The abstract before semantic domain $\hat{D}_t$, the domain $\hat{E}_t$ of abstract before environments are defined as follows:

$$\hat{D}_t = \sum_{\tau} \hat{D}_t^{\tau} \quad \text{/* Abstract before semantic domain */}$$
$$\hat{E}_t = Id \rightarrow \hat{D}_t \quad \text{/* Domain of abstract before environments */}$$

The before subdomain $\hat{D}_t^{\tau}$ for expressions of type τ is defined as follows:

$$\hat{D}_t^{int} = \hat{B}_t \times \{err\} \quad \text{abstract subdomain for integers}$$
$$\hat{D}_t^{bool} = \hat{B}_t \times \{err\} \quad \text{abstract subdomain for booleans}$$
$$\hat{D}_t^{\tau_1 \rightarrow \tau_2} = \hat{B}_t \times (\hat{D}_t^{\tau_1} \rightarrow \hat{D}_t^{\tau_2}) \quad \text{abstract subdomain for functions of type } \tau_1 \rightarrow \tau_2$$

Abstract Before Semantic Functions

The abstract before semantic functions are defined as follows:

$$\hat{T}_c : Con \rightarrow \hat{D}_t \quad \text{/* Abstract before semantic function for constants */}$$
$$\hat{T}_e : Exp \rightarrow \hat{E}_t \rightarrow \hat{D}_t \quad \text{/* Abstract before semantic function for expressions */}$$
$$\hat{T}_{pr} : Program \rightarrow \hat{D}_t \quad \text{/* Abstract before semantic function for programs */}$$

The abstract before semantic functions are given in Figure 5.4. $e\hat{n}v_t$ is any abstract before environment in $\hat{E}_t$, and $\hat{nullenv}_t$ is an abstract before environment that maps every identifier to the least element of its abstract before semantic domain.

Safety and Termination

We introduce the notion of *safety* which relates the exact before semantics to the abstract before semantics. Let u and v be values of an expression e of type τ or τ *list* in $\hat{D}_t$ and D_t, respectively. Let n be the number of arguments that the type τ can take before returning a value of primitive type. We say that u is a *safe* approximation (with respect to before information) of v iff

$$(\mathrm{NAP}_k(v, s_1, \ldots, s_k))_{(1)} \sqsubseteq (\mathrm{NAP}_k(u, t_1, \ldots, t_k))_{(1)}.$$

for all $k \leq n$ where s_i is a *safe* approximation of t_i for all $i \leq k$.

Theorem 5.1 (Safety) *For any expression e, and environments env_t and $e\hat{n}v_t$ such that for all y, $e\hat{n}v_t[\![y]\!]$ is safe for $env_t[\![y]\!]$, $\hat{T}_e[\![e]\!]e\hat{n}v_t$ is safe (with respect to before information) for $T_e[\![e]\!]env_t$. Thus, the before information obtained by the exact before semantics implies the before information obtained by the abstract before semantics.*

Proof : We can prove by structural induction on expression e.
I. Base Case:

$$\hat{T}_c[\![c]\!] \;=\; \langle 1, err\rangle \quad c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\}$$

$$\hat{T}_e[\![c]\!]e\hat{n}v_t \;=\; \hat{T}_c[\![c]\!]$$

$$\hat{T}_e[\![x]\!]e\hat{n}v_t \;=\; e\hat{n}v_t[\![x]\!]$$

$$\begin{array}{lcll}\hat{T}_e[\![e_1 + e_2]\!]e\hat{n}v_t & = & \text{let} & \hat{l} = \hat{T}_e[\![e_1]\!]e\hat{n}v_t \qquad /^*\text{ same for } e_1 - e_2 \text{ and } e_1 = e_2\ ^*/\\ & & & \hat{r} = \hat{T}_e[\![e_2]\!]e\hat{n}v_t\\ & & \text{in} & \langle \hat{l}_{(1)} \rhd \hat{r}_{(1)}, err\rangle\end{array}$$

$$\begin{array}{lcll}\hat{T}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\hat{n}v_t & = & \text{let} & \hat{p} = \hat{T}_e[\![e_1]\!]e\hat{n}v_t\\ & & & \hat{c} = \hat{T}_e[\![e_2]\!]e\hat{n}v_t\\ & & & \hat{a} = \hat{T}_e[\![e_3]\!]e\hat{n}v_t\\ & & \text{in} & \langle \hat{p}_{(1)} \rhd \hat{c}_{(1)}, \hat{c}_{(2)}\rangle \bigsqcup \langle \hat{p}_{(1)} \rhd \hat{a}_{(1)}, \hat{a}_{(2)}\rangle\end{array}$$

$$\begin{array}{lcll}\hat{T}_e[\![e_1 e_2]\!]e\hat{n}v_t & = & \text{let} & \hat{f} = \hat{T}_e[\![e_1]\!]e\hat{n}v_t)\\ & & & \hat{ap} = \hat{f}_{(2)}(\hat{T}_e[\![e_2]\!]e\hat{n}v_t)\\ & & \text{in} & \langle \hat{f}_{(1)} \rhd \hat{ap}_{(1)}, \hat{ap}_{(2)}\rangle\end{array}$$

$$\hat{T}_e[\![\texttt{lambda}(x).e]\!]e\hat{n}v_t \;=\; \langle 1, \lambda y.\hat{T}_e[\![e]\!]e\hat{n}v_t[x \mapsto y]\rangle$$

$$\hat{T}_e[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n; \texttt{in } e]\!]e\hat{n}v_t \;=\; \hat{T}_e[\![e]\!]e\hat{n}v'_t$$

where $e\hat{n}v'_t = e\hat{n}v_t[x_1 \mapsto \hat{T}_e[\![e_1]\!]e\hat{n}v'_t, \ldots, x_n \mapsto \hat{T}_e[\![e_n]\!]e\hat{n}v'_t]$

$$\hat{T}_{pr}[\![pr]\!] \;=\; \hat{T}_e[\![pr]\!]\hat{nullenv}_t$$

Figure 5.4: Abstract Before Semantic Functions

1. $e = c$: $\hat{T}_e[\![c]\!]e\hat{n}v_t = \hat{T}_c[\![c]\!]$ and $T_e[\![c]\!]env_t = T_c[\![c]\!]$. For $c \in \{\ldots,$`-1,0,1,...,` `true, false, nil, null` $\}$, $T_c[\![c]\!] = \hat{T}_c[\![c]\!]$ and then it clearly holds.

2. $e = x$: $\hat{T}_e[\![x]\!]e\hat{n}v_t = e\hat{n}v_t[\![x]\!]$ and $T_e[\![x]\!]env_t = env_t[\![x]\!]$. Since, for all y, $e\hat{n}v_t[\![y]\!]$ is safe for $env_t[\![y]\!]$, it clearly holds.

II. Structural Induction Step: Assume that $\hat{T}_e[\![e]\!]e\hat{n}v_t$ is safe for $T_e[\![e]\!]env_t$ for expressions such as e_0, e_1, e_2, e_3 and e_n. (structural induction hypothesis)

1. $e = e_1 + e_2$: $\hat{T}_e[\![e_1+e_2]\!]e\hat{n}v_t = \langle \hat{l}_{(1)} \triangleright \hat{r}_{(1)}, err\rangle$ where $\hat{l} = \hat{T}_e[\![e_1]\!]e\hat{n}v_t$ and $\hat{r} = \hat{T}_e[\![e_2]\!]e\hat{n}v_t$. $T_e[\![e_1+e_2]\!]env_t = \langle l_{(1)} \triangleright r_{(1)}, err\rangle$ where $l = T_e[\![e_1]\!]env_t$ and $r = T_e[\![e_2]\!]env_t$. By the structural induction hypothesis, $\hat{l}$ and $\hat{r}$ are safe for l and r, respectively. Thus, it holds. Similarly, it holds for $e_1 - e_2$ and $e_1 = e_2$.

2. $e =$ `if` e_1 `then` e_2 `else` e_3: $\hat{T}_e[\![$`if` e_1 `then` e_2 `else` $e_3]\!]e\hat{n}v_t = \langle \hat{p}_{(1)} \triangleright \hat{c}_{(1)}, \hat{c}_{(2)}\rangle \sqcup \langle \hat{p}_{(1)} \triangleright \hat{a}_{(1)}, \hat{c}_{(2)}\rangle$ where $\hat{p} = \hat{T}_e[\![e_1]\!]e\hat{n}v_t$, $\hat{c} = \hat{T}_e[\![e_2]\!]e\hat{n}v_t$, and $\hat{a} = \hat{T}_e[\![e_3]\!]e\hat{n}v_t$. $T_e[\![$`if` e_1 `then` e_2 `else` $e_3]\!]env_t$ is either $\langle p_{(1)} \triangleright c_{(1)}, c_{(2)}\rangle$ or $\langle p_{(1)} \triangleright c_{(1)}, c_{(2)}\rangle$ where $p = \hat{T}_e[\![e_1]\!]e\hat{n}v_t$, $c = \hat{T}_e[\![e_2]\!]e\hat{n}v_t$, and $a = \hat{T}_e[\![e_3]\!]e\hat{n}v_t$ depending on the standard semantic value of $\mathtt{e}_1$. In any case, by the structural induction hypothesis and the definition of $\triangleright$, it holds.

3. $e = e_1e_2$: $\hat{T}_e[\![e_1e_2]\!]e\hat{n}v_t = \langle \hat{f}_{(1)} \triangleright \hat{ap}_{(1)}, \hat{ap}_{(2)}\rangle$ where $\hat{f} = \hat{T}_e[\![e_1]\!]e\hat{n}v_t)$, and $\hat{ap} = \hat{f}_{(2)}(\hat{T}_e[\![e_2]\!]e\hat{n}v_t)$. $T_e[\![e_1e_2]\!]env_t = \langle f_{(1)} \triangleright ap_{(1)}, ap_{(2)}\rangle$ where $f = \hat{T}_e[\![e_1]\!]e\hat{n}v_t)$, and $ap = f_{(2)}(\hat{T}_e[\![e_2]\!]e\hat{n}v_t)$. By the structural induction hypothesis, $\hat{f}$ and $\hat{ap}$ are safe for f and ap, respectively. Then, by the definition of $\triangleright$ and safe, it holds.

4. $e =$ `lambda`$(x).e_1$: $\hat{T}_e[\![$`lambda`$(x).e]\!]e\hat{n}v_t = \langle 1, \lambda y.\hat{T}_e[\![e]\!]e\hat{n}v_t[x \mapsto y]\rangle$. $T_e[\![$`lambda`$(x).e]\!]$ $env_t = \langle 1, \lambda y.T_e[\![e]\!]env_t[x \mapsto y]\rangle$. By the structural induction hypothesis, $\hat{T}_e[\![e]\!]e\hat{n}v_t[x \mapsto y]$ is safe for $T_e[\![e]\!]env_t[x \mapsto y]$ and thus it holds.

5. $e =$ `letrec` $x_1 = e_1; \ldots; x_n = e_n;$ `in` e_0: $\hat{T}_e[\![$`letrec` $x_1 = e_1; \ldots;\ x_n = e_n;$ `in` $e_0]\!]e\hat{n}v_t = \hat{T}_e[\![e_0]\!]e\hat{n}v'_t$ where $e\hat{n}v'_t = e\hat{n}v_t[x_i \mapsto \hat{T}_e[\![e_i]\!]e\hat{n}v'_t]$, and $T_e[\![$`letrec` $x_1 = e_1; \ldots;\ x_n = e_n;$ `in` $e_0]\!]$ $env_t = T_e[\![e_0]\!]env'_t$ where $env'_t = env_t[x_i \mapsto T_e[\![e_i]\!]env'_t]$. Here, $e\hat{n}v'$ and env' are recursively defined. We prove that $e\hat{n}v'_t[\![y]\!]$ is safe for $env'_t[\![y]\!]$ for all y by fixpoint induction on environments env'.

   1. Base Case: The first approximation $env_t^{'(0)}$ of $e\hat{n}v'_t$ is $e\hat{n}v_t[x_i \mapsto \perp]$. The first approximation $env_t^{'(0)}$ of env'_t is $env_t[x_i \mapsto \perp]$. Thus, for all y, $e\hat{n}v'_t[\![y]\!]$ is safe for $env'_t[\![y]\!]$. Then, by the structural induction hypothesis, $\hat{T}_e[\![e_0]\!]e\hat{n}v'_t$. is safe for $T_e[\![e_0]\!]env'_t$.

   2. Fixpoint Induction Step: Assume that, for some fixed $k \geq 0$, the k^{th} approximation $e\hat{n}v_t^{'(k)}[\![y]\!]$ is safe for $env_t^{'(k)}[\![y]\!]$ for all y. (fixpoint induction hypothesis) Then, The $(k+1)^{th}$ approximation $e\hat{n}v_t^{'(k+1)} = e\hat{n}v_t[x_i \mapsto \hat{T}_e[\![e_i]\!]e\hat{n}v_t^{'(k)}]$, and $env_t^{'(k+1)} = env_t[x_i \mapsto T_e[\![e_i]\!]env_t^{'(k)}]$. By the structural induction hypothesis, $\hat{T}_e[\![e_i]\!]e\hat{n}v_t^{'(k)}$ is safe

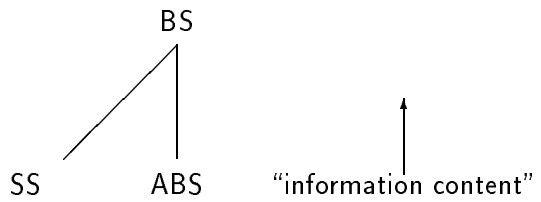

Figure 5.5: Relationship among Standard and Before semantics

for $T_e[\![e_i]\!]env_t^{'(k)}$. Thus, $e\hat{n}v_t^{'(k+1)}[\![y]\!]$ is safe for $env_t^{'(k+1)}[\![y]\!]$ for all y. Then, by the structural induction hypothesis, $\hat{T}_e[\![e_0]\!]e\hat{n}v_t'$ is safe for $T_e[\![e_0]\!]env_t'$.

□

Theorem 5.2 (Termination) *For any (finite) program pr* $\in$ *Program,* $\hat{T}_{pr}[\![pr]\!]$ *is computable, i.e. it always terminates in finite number of steps.*

Proof : All the abstract before semantic domains are finite, and, because the operator $\triangleright$ is monotonic, all functions over before semantics domains are monotonic functions. Thus, the fixpoints can be computed in a finite number of steps, and the interpretation under the abstract before semantics is guaranteed to terminate for all programs. □

The relationship among the standard semantics (SS), the non-standard exact before semantics (BS), and the abstract before semantics (ABS) is shown in Figure 5.5.

5.2.3 Testing for Before Demand

Since the abstract before semantics is guaranteed to terminate, the abstract before semantics can be used as a basis to infer the before information at compile-time. We consider two kinds of before information for higher-order functions: global and local. The global before information of a function f defined by $f\ x_1\ \ldots x_n = body_f$ holds true for *every possible application* of f to n arguments, and is determined by examining only at the body of f. The local before information is computed for a *particular application* of f and depends upon the number and properties of the arguments in the application. It turns out that the global before information is equivalent to the local before information in case of first-order functions.

Global Before Test

The global before testing of a function f is performed by applying the abstract before semantic value of f to n arguments in $\hat{D}_t$ that contain the least before information possible. Thus, any before information obtained from the application arises solely from properties of f.

Definition 5.2 (Worst-case Before Function) For each type τ such that m is the number of arguments that a function of type τ can take before returning a primitive value, we define the abstract function W^τ that corresponds to the worst-case (i.e. least) before information.

$$W^\tau \stackrel{\text{def}}{=} \begin{cases} \lambda x_1.\langle 1, \lambda x_2.\langle 1, \ldots, \lambda x_m.\langle \bigsqcup_{i=1}^{m} x_{i(1)}, err\rangle \ldots\rangle\rangle & m \geq 1 \\ err & m = 0 \end{cases}$$

Given a function f defined by $f\ x_1\ \ldots\ x_n = body_f$ of type $f : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$, two sets X and Y of occurrences of the formal parameters of f, and an abstract before semantic environment $e\hat{n}v_t$ mapping f to an element in $\hat{D}_t$, the global analysis function G_before? determines whether the before relation between X and Y holds globally. It is defined as follows:

$$\begin{aligned} &\mathsf{G_before?}(f, X, Y, e\hat{n}v_t) = \\ &\qquad \hat{T}_e[\![f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots x_{n1}\ldots x_{no(n)}]\!]e\hat{n}v_t[f' \mapsto \hat{f}', x_{ij} \mapsto y_{ij}] \end{aligned}$$

where

f' is the auxiliary function for f, $\hat{f}' = \hat{T}_e[\![f']\!]e\hat{n}v_t$, $o(i)$ is the number of occurrences of x_i,

for each $x_{ij} \in X$,

$y_{ij} = \langle 2, W^{\tau_i}\rangle$,

for each $x_{ij} \in Y$,

$y_{ij} = \langle 0, W^{\tau_i}\rangle$,

and for each $x_{ij} \notin X \cup Y$,

$x_{ij} = \langle 1, W^{\tau_i}\rangle$.

The value that G_before?$(f, X, Y, e\hat{n}v_t)$ returns is a pair $\langle \hat{p}, \hat{g}\rangle$ interpreted as follows:

- If $\hat{p} = 0$ or 1, then we can conclude that $Y \prec X$ in any possible application of f to n arguments.
- Otherwise, if $\hat{p} = 2$ then an occurrence in Y might or might not be demanded before the first demand of any occurrence in X. Thus, in this case, we cannot conclude that $Y \prec X$ in all possible applications of f.

If the result in the standard semantics of the application of f to n arguments is a function g, then G_before? will only give us the order-of-demand information about the bound variable occurrences inside the body of f during the execution of the application of f. We might also want to obtain order-of-demand information about the bound variable occurrences inside f if g were subsequently applied to other arguments. This extra amount of order-of-demand information is obtained by applying $\hat{g}$ to the abstract value $\langle 1, W^{\tau_g} \rangle$, where τ_g is the type of the argument expected by the result of f.

For example, consider a function defined by

```
f x y z = if x = 0 then lambda(w).z+y+w
          else lambda(w).z*y*w
```

The auxiliary function `f'` derived from `f` is

```
f' x y1 y2 z1 z2 = if x = 0 then lambda(w).z1+y1+w1
                            else lambda(w).z2*y2*w2
```

G_before?(`f`,{`y1`,`y2`},{`z1`,`z2`}) would return $< 1, \hat{g} >$ indicating that neither `y` nor `z` is demanded during an application of `f` to three arguments. If the result of the application of `f` is applied to a fourth argument, the information that `z` would always be demanded before `y` can be obtained by applying $\hat{g}$ to the abstract value $\langle 1, err \rangle$.

Local Before Test

Given a function f defined by $f\ x_1\ \ldots\ x_n = body_f$ of type $f : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ in a particular application $f\ e_1\ \ldots\ e_m$, two sets X and Y of occurrences of f's formal parameters, and an abstract before semantic environment $e\hat{n}v_t$ mapping f and all free identifiers within e_1 and e_n to elements of $\hat{D}_t$, the local analysis function L_before? determines whether the before relation between X and Y holds locally in $f\ e_1\ \ldots\ e_m$. It is defined as follows:

$$\text{L_before?}(f, e_1, \ldots, e_m, X, Y, e\hat{n}v_t) =$$
$$(\hat{T}_e[\![f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots x_{m1} \ldots x_{mo(m)}]\!]e\hat{n}v_t[f' \mapsto \hat{f}', x_{ij} \mapsto x_{ij}])_{(1)}$$

where

f' is the auxiliary function for f, $\hat{f}' = \hat{T}_e[\![f']\!]e\hat{n}v_t$, $o(i)$ is the number of occurrences of x_i,

for each $x_{ij} \in X$,

$$y_{ij} = \langle 2, (\hat{T}_e[\![e_i]\!]e\hat{n}v_t)_{(2)} \rangle,$$

for each $x_{ij} \in Y$,

$$y_{ij} = \langle 0, (\hat{T}_e[\![e_i]\!]e\hat{n}v_t)_{(2)} \rangle,$$

and for each $x_{ij} \notin X \cup Y$,

$$y_{ij} = \langle 1, (\hat{T}_e[\![e_i]\!]e\hat{n}v_t)_{(2)} \rangle.$$

Then,

- If L_before?$(f, e_1, \ldots, e_m, X, Y, e\hat{n}v_t) = 0$ or 1 then we can conclude that $Y \prec X$ in the evaluation of $(f\ e_1\ \ldots e_m)$.
- Otherwise, if L_before?$(f, e_1, \ldots, e_m, X, Y, e\hat{n}v_t) = 2$ then an occurrence in Y might or might not be demanded before an occurrence in X. Thus, in this case, we are unable to conclude that $Y \prec X$ during the evaluation of $(f\ e_1\ \ldots\ e_m)$.

5.3 Using Before Analysis

Our motivation for developing the order-of-demand analysis is to obtain evaluation status information. As we describe above, evaluation status information can be used to optimize lazy evaluation by avoiding run-time checks to see if a bound variable is evaluated or not. Since each occurrence of a bound variable corresponds to a potential demand for the value of the variable, order-of-demand analysis is used to determine which occurrences correspond to demands of an already evaluated variable.

As a nice side-effect, order-of-demand analysis can be used to compute (approximately, of course) the relative evaluation orders of the parameters of a function, and the strictness properties of a function.

5.3.1 Computing Status-of-Evaluation Information

Given an occurrence x_{ij} of a bound variable x_i, if we can determine that there will always be some other occurrence x_{ik} of x_i demanded before x_{ij}, then no run-time test of the status of x_i is required when x_{ij} is demanded.

We can determine the evaluation status of x_{ij} when it is demanded by computing the order of demand information between x_{ij} and all other occurrences of x_i.

Theorem 5.3 *Let f be a function defined by $f\ x_1 \ldots x_n = body_f$ and $x_{i1}, \ldots, x_{io(i)}$ be all the occurrences of x_i in $body_f$. Then,*

1. $(\mathsf{G_before?}(f, \{x_{ij}\}, \{x_{i1}, \ldots, x_{io(i)}\} - \{x_{ij}\}, e\hat{n}v_t))_{(1)} = 0$ *or* $1 \Longrightarrow$ *whenever x_{ij} is demanded, x_i will have already been evaluated and thus the demand can be satisfied by retrieving the saved value, for any possible application of f.*

2. $(\mathsf{G_before?}(f, \{x_{i1}, \ldots, x_{io(i)}\} - \{x_{ij}\}, \{x_{ij}\}, e\hat{n}v_t))_{(1)} = 0$ *or* $1 \Longrightarrow$ *the demand on x_{ij} the first demand of any occurrence of x_i, and thus will always be the occurrence (if any) causing the evaluation of x_i.*

Proof : If $(\mathsf{G_before?}(f, \{x_{ij}\}, \{x_{i1}, \ldots, x_{io(i)}\} - \{x_{ij}\}), e)_{(1)} = 0$ or 1 then some occurrence of x_i other than x_{ij} is always demanded before the occurrence x_{ij} is ever demanded first. This means that whenever x_{ij} is ever demanded, x_i has been already evaluated.

If $(\mathsf{G_before?}(f, \{x_{i1}, \ldots, x_{io(i)}\} - \{x_{ij}\}, \{x_{ij}\}, e\hat{n}v_t))_{(1)} = 0$ or 1 then before any occurrence x_{ik} of x_i, $k \neq j$, is demanded x_{ij} must be demanded. Thus, whenever x_{ij} is ever demanded it is the first demand of x_i which causes actual evaluation of x_i, and thus x_{ij} has an unevaluated status. □

A similar theorem can be stated about the use of the local *before* analysis to provide evaluation status information about a particular function application.

5.3.2 Computing Order-of-Evaluation Information

In lazy evaluation, the evaluation of an argument takes place only when some occurrence of the corresponding parameter is demanded. The order of evaluation between two parameters of a function can be determined by computing the order of demands between the set of occurrences of one parameter and the set of occurrences of the other.

Theorem 5.4 *Let f be a function defined by $f\ x_1 \ldots x_n = body_f$, and let $x_{i1}, \ldots, x_{io(i)}$ and $x_{j1}, \ldots, x_{jo(j)}$ be all of the occurrences of x_i and x_j in $body_f$, respectively. Then,*

1. $(\mathsf{G_before?}(f, \{x_{i1}, \ldots, x_{io(i)}\}, \{x_{j1}, \ldots, x_{jo(j)}\}, e\hat{n}v_t))_{(1)}$ = *0 or 1* $\Longrightarrow$ x_j *is evaluated before* x_i *in any possible application of* f.

Proof : If $(\mathsf{G_before?}(f, \{x_{i1}, \ldots, x_{io(i)}\}, \{x_{j1}, \ldots, x_{jo(j)}\}, e\hat{n}v_t))_{(1)}$= 0 or 1 then before any demand of an occurrence x_{ik} of x_i, there is a demand on an occurrence x_{jm} of x_j. Thus, $x_j$is always evaluated before x_i.

□

A similar theorem can be stated about the use of the local *before* analysis to provide evaluation order information for the arguments in a particular function application.

It is important to note that although it is clearly not necessary to distinguish the individual occurrences of each formal parameter to perform an order-of-evaluation analysis, there is no added cost in doing so (other than simply labeling the occurrences). Since all occurrences of a parameter are placed in the same set, they are treated as a unit by our order-of-demand analysis. The cost is not proportional to the size of the sets of occurrences, but rather to the number of times the order-of-demand analysis must be performed on different sets.

5.3.3 Computing Strictness Information

Strictness information about a function, both in any possible application of the function and in a particular application of the function, can be determined using order-of-demand analysis. Like all compile-time strictness analyses, the order-of-demand analysis provides a safe *approximation* of the actual strictness properties of a function.

Theorem 5.5 *Let* f *be a function defined by* $f\ x_1 \ldots x_n = body_f$ *and let* $x_{i1}, \ldots, x_{io(i)}$ *be all occurrences of a single parameter* x_i *in* $body_f$. *Then,*

1. $(\mathsf{G_before?}(f, \{\$\}, \{x_{i1}, \ldots, x_{io(i)}\}, e\hat{n}v_t))_{(1)} = 0 \Longrightarrow$ f *is strict in* x_i, *in any possible application of* f.
2. $\mathsf{L_before?}(f, e_1, \ldots, e_n, \{\$\}, \{x_{i1}, \ldots, x_{io(i)}\}, e\hat{n}v_t) = 0 \Longrightarrow$ f *is strict in* x_i *in the particular application of* f *to* e_1 *through* e_n.

Proof : We sketch the proof here for the global strictness analysis. The proof for the local strictness analysis is similar. $(\mathsf{G_before?}(f, \{\$\}, \{x_{i1}, \ldots, x_{io(i)}\}, e\hat{n}v_t))_{(1)} = 0$ implies one of two things:

1. At least one of $x_{i1} \ldots x_{io(i)}$ is demanded during the any application of f to n arguments. This is obtained from the meaning of 0 in $\hat{B}_{\prec}$, that is $\{x_{i1} \ldots x_{io(i)}\} \prec \{\$\}$.

2. Or, because the bottom element of $\hat{B}_{\prec}$ is 0, the fixpoint finding iteration determined that the fixpoint of f' is a function that always return 0, which would be the case if f' (and thus f) is everywhere undefined (non-terminating).

In either case, f is strict with respect to x_i. □

5.3.4 Examples

Consider the following lazy functional program:

```
letrec f w x y z = if (w=0) then z+x elseif (y=0)
                   then z+x else (z+y) + f (w-1) x (y-1) z
in  ...
```

We assume that `f` is of type $int \to int \to int \to int \to int$. The definition of the before semantic value f of `f` is:

$$\begin{aligned} f\ w\ x\ y\ z &= \langle (w_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup (w_{(1)} \rhd (y_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup \\ &\quad (y_{(1)} \rhd z_{(1)} \rhd y_{(1)} \rhd (f\ \langle w_{(1)} \rhd 1, err\rangle\ x\ \langle y_{(1)} \rhd 1, err\rangle\ z)_{(1)})), err\rangle \\ &= \langle (w_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup \\ &\quad (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd (f\ \langle w_{(1)} \rhd 1, err\rangle\ x\ \langle y_{(1)} \rhd 1, err\rangle\ z)_{(1)})), err\rangle \end{aligned}$$

Since f is recursively defined, it can be computed in a finite steps using a fixpoint finding method as follows:

$$\begin{aligned} f^{(0)}\ w\ x\ y\ z &= \langle 0, err\rangle \\ f^{(1)}\ w\ x\ y\ z &= \langle (w_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup \\ &\quad (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd 0), err\rangle \\ f^{(2)}\ w\ x\ y\ z &= \langle (w_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup \\ &\quad (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd 0), err\rangle \end{aligned}$$

Since $f^{(1)} = f^{(2)}$, we have that

$$\begin{aligned} f\ w\ x\ y\ z &= \langle 1, \lambda w.\langle 1, \lambda x.\langle 1, \lambda y.\langle 1, \lambda z.\langle (w_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup \\ &\quad (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd x_{(1)}) \sqcup (w_{(1)} \rhd y_{(1)} \rhd z_{(1)} \rhd 0), err\rangle\rangle\rangle\rangle\rangle \end{aligned}$$

The auxiliary function `f'` of `f` is defined as follows:

```
f' w1 w2 x1 x2 x3 y1 y2 y3 z1 z2 z3 z4 = if (w1=0) then z1+x1
                                elseif (y1=0) then z2+x2
                                else (z3+y2) + f (w2-1) x3 (y3-1) z4
```

Again, notice that `f'` is not recursive, but rather calls `f`. Then, the definition of the before semantic value f' of `f'` is given as follows (without a fixpoint iteration).

$$
\begin{aligned}
&f'\ w_1\ w_2\ x_1\ x_2\ x_3\ y_1\ y_2\ y_3\ z_1\ z_2\ z_3\ z_4 = \\
&\quad \langle 1, \lambda w_1.\langle 1, \lambda w_2.\langle 1, \lambda x_1.\langle 1, \lambda x_2.\langle 1, \lambda x_3.\langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \lambda y_3. \\
&\quad \langle 1, \lambda z_1.\langle 1, \lambda z_2.\langle 1, \lambda z_3.\langle 1, \lambda z_4.\langle (w_{1(1)} \rhd z_{1(1)} \rhd x_{1(1)}) \sqcup \\
&\quad (w_{1(1)} \rhd y_{1(1)} \rhd z_{2(1)} \rhd x_{2(1)}) \sqcup (w_{1(1)} \rhd y_{1(1)} \rhd z_{3(1)} \rhd y_{2(1)} \rhd \\
&\quad (f\ \langle w_{2(1)} \rhd 1, err\rangle\ x_3\ \langle y_{3(1)} \rhd 1, err\rangle\ z_4)_{(1)}), err\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle
\end{aligned}
$$

Evaluation-Status Information

Let $\hat{env}_t = [\mathtt{f} \mapsto f]$.

$$
\begin{aligned}
&(\mathsf{G_before?}(\mathtt{f}, \{\mathtt{y2}\}, \{\mathtt{y1}, \mathtt{y3}\}, \hat{env}_t))_{(1)} = \\
&\quad \hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]\hat{env'}_t)_{(1)} = 1
\end{aligned}
$$

where

$$
\begin{aligned}
\hat{env'}_t \ = \ & \hat{env}_t[\mathtt{f'} \mapsto f', \mathtt{y2} \mapsto \langle 2, err\rangle, \mathtt{y1}, \mathtt{y3} \mapsto \langle 0, err\rangle, \\
& \mathtt{w1}, \mathtt{w2}, \mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4} \mapsto \langle 1, err\rangle]
\end{aligned}
$$

Thus, we can conclude that when `y2` is demanded, either `y1` or `y3` has already been evaluated and no run-time check is required. Similarly, we conclude that when `y3` is demanded, either `y1` or `y2` has already been already evaluated because

$$
\begin{aligned}
&(\mathsf{G_before?}(\mathtt{f}, \{\mathtt{y3}\}, \{\mathtt{y1}, \mathtt{y2}\}, \hat{env}_t))_{(1)} = \\
&\quad \hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]\hat{env'}_t)_{(1)} = 1
\end{aligned}
$$

where

$$
\begin{aligned}
\hat{env'}_t \ = \ & \hat{env}_t[\mathtt{f'} \mapsto f', \mathtt{y3} \mapsto \langle 2, err\rangle, \mathtt{y1}, \mathtt{y2} \mapsto \langle 0, err\rangle, \\
& \mathtt{w1}, \mathtt{w2}, \mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4} \mapsto \langle 1, err\rangle]
\end{aligned}
$$

And, because

$$
\begin{aligned}
&(\mathsf{G_before?}(\mathtt{f}, \{\mathtt{y2}, \mathtt{y3}\}, \{\mathtt{y1}\}, \hat{env}_t))_{(1)} = \\
&\quad \hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]\hat{env'}_t)_{(1)} = 1
\end{aligned}
$$

where

$$
\begin{aligned}
\hat{env'}_t \ = \ & \hat{env}_t[\mathtt{f'} \mapsto f', \mathtt{y2}, \mathtt{y3} \mapsto \langle 2, err\rangle, \mathtt{y1} \mapsto \langle 0, err\rangle, \\
& \mathtt{w1}, \mathtt{w2}, \mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4} \mapsto \langle 1, err\rangle]
\end{aligned}
$$

we can conclude that `y1` is the occurrence of `y` that is demanded first within `f`.

Evaluation-Order Information

$$(\mathsf{G_before?}(\mathtt{f}, \{\mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4}\}, \{\mathtt{w1}, \mathtt{w2}\}, e\hat{n}v_t))_{(1)} =$$
$$\hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]e\hat{n}v'_t)_{(1)} = 0$$

where

$$e\hat{n}v'_t = e\hat{n}v_t[\mathtt{f'} \mapsto f', \mathtt{w1}, \mathtt{w2} \mapsto \langle 0, err\rangle, \mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4} \mapsto \langle 2, err\rangle,$$
$$\mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{y1}, \mathtt{y2}, \mathtt{y3} \mapsto \langle 1, err\rangle]$$

Thus, we can conclude that `w` is always evaluated before `z` in `f`.

Similarly, we also conclude that `z` is always evaluated before `x` because

$$(\mathsf{G_before?}(\mathtt{f}, \{\mathtt{x1}, \mathtt{x2}, \mathtt{x3}\}, \{\mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4}\}, e\hat{n}v_t))_{(1)} =$$
$$\hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]e\hat{n}v'_t)_{(1)} = 0$$

where

$$e\hat{n}v'_t = e\hat{n}v_t[\mathtt{f'} \mapsto f', \mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4} \mapsto \langle 0, err\rangle, \mathtt{x1}, \mathtt{x2}, \mathtt{x3} \mapsto \langle 2, err\rangle,$$
$$\mathtt{y1}, \mathtt{y2}, \mathtt{y3} \mapsto \langle 1, err\rangle]$$

Hence, we conclude that the parameters to `f` are evaluated in the order `w, z, x`. In addition, we can conclude that `w` is evaluated before `y`, if `y` is evaluated at all. We cannot conclude anything, however, about the relative evaluation order between `x` and `y` and between `z` and `y`.

Strictness Information

$$(\mathsf{G_before?}(\mathtt{f}, \{\$\}, \{\mathtt{w1}, \mathtt{w2}\}, e\hat{n}v_t))_{(1)} =$$
$$\hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]e\hat{n}v'_t)_{(1)} = 0$$

where

$$e\hat{n}v'_t = e\hat{n}v_t[\mathtt{f'} \mapsto f', \mathtt{w1}, \mathtt{w2} \mapsto \langle 0, err\rangle, \mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{y1}, \mathtt{y2}, \mathtt{y3}, \mathtt{z1}, \mathtt{z2}, \mathtt{z3}, \mathtt{z4} \mapsto$$
$$\langle 1, err\rangle]$$

Then, we conclude that `f` is strict in `w`. Similarly, we conclude that `f` is strict in `x` and `z`, because

$$(\mathsf{G_before?}(\mathtt{f}, \{\$\}, \{\mathtt{x1}, \mathtt{x2}, \mathtt{x3}\}, e\hat{n}v_t))_{(1)} =$$
$$\hat{T}_e[\![\mathtt{f'\ w1\ w2\ x1\ x2\ x3\ y1\ y2\ y3\ z1\ z2\ z3\ z4}]\!]e\hat{n}v'_t)_{(1)} = 0$$

where

$$\hat{env'_t} = \hat{env_t}[\texttt{f'} \mapsto f', \texttt{x1}, \texttt{x2}, \texttt{x3} \mapsto \langle 0, err\rangle, \texttt{w1}, \texttt{w2}, \texttt{y1}, \texttt{y2}, \texttt{y3}, \texttt{z1}, \texttt{z2}, \texttt{z3}, \texttt{z4} \mapsto \langle 1, err\rangle]$$

and

$$(\mathsf{G_before?}(\texttt{f}, \{\$\}, \{\texttt{z1}, \texttt{z2}, \texttt{z3}, \texttt{z4}\}, \hat{env_t}))_{(1)} = \hat{T_e}[\![\texttt{f' w1 w2 x1 x2 x3 y1 y2 y3 z1 z2 z3 z4}]\!]\hat{env'_t})_{(1)} = 0$$

where

$$\hat{env'_t} = \hat{env_t}[\texttt{f'} \mapsto f', \texttt{z1}, \texttt{z2}, \texttt{z3}, \texttt{z4} \mapsto \langle 0, err\rangle, \texttt{w1}, \texttt{w2}, \texttt{x1}, \texttt{x2}, \texttt{x3}, \texttt{y1}, \texttt{y2}, \texttt{y3} \mapsto \langle 1, err\rangle]$$

However, since

$$(\mathsf{G_before?}(\texttt{f}, \{\$\}, \{\texttt{y1}, \texttt{y2}, \texttt{y3}\}, \hat{env_t}))_{(1)} = \hat{T_e}[\![\texttt{f' w1 w2 x1 x2 x3 y1 y2 y3 z1 z2 z3 z4}]\!]\hat{env'_t})_{(1)} = 1$$

where

$$\hat{env'_t} = \hat{env_t}[\texttt{f'} \mapsto f', \texttt{y1}, \texttt{y2}, \texttt{y3} \mapsto \langle 0, err\rangle, \texttt{w1}, \texttt{w2}, \texttt{x1}, \texttt{x2}, \texttt{x3}, \texttt{z1}, \texttt{z2}, \texttt{z3}, \texttt{z4} \mapsto \langle 1, err\rangle]$$

we cannot conclude that `f` is strict in `y`.

5.4 Complexity of Before Analysis

The abstract interpretation framework for order-of-demand analysis is very similar to the framework for strictness analysis except that a three-element (rather than two-element) domain is used as the basic abstract semantic domain. Thus, the order of complexity of order-of-demand analysis is that of strictness analysis which is known to be exponential in the worst-case. The majority of the complexity of analysis based on the abstract interpretation technique comes from finding the fixpoints of recursive functions in the abstract semantic domains. Advanced methods for finding fixpoints on finite domains can also be applied to order-of-demand analysis in order to reduce the average-case complexity. The complexity of finding fixpoints depends on the size of the abstract semantic domains. It has been argued that for many function, especially in the higher-order case, finding fixpoints is intractable unless the sizes of the abstract domains are reduced. Our method has a much lower complexity than the path model [11], even in the first-order case.

5.5 Extensions to Other Evaluation Models

So far we have described the order-of-demand analysis for a sequential lazy evaluation model. In this section, we discuss various extensions to parallel lazy evaluation, to optimized lazy evaluation using strictness information, and to non-flat domains.

Consider a parallel lazy evaluation model in which arguments of primitive functions may be evaluated in parallel. The semantic equations of the abstract before semantic function for strict primitive functions are modified as follows:

$$\begin{array}{rcll} \hat{T}_e[\![e_1 + e_2]\!]e\hat{n}v_t & = & \text{let} & l = \hat{T}_e[\![e_1]\!]e\hat{n}v_t /^* \text{ same for } e_1 - e_2 \text{ and } e_1 = e_2 \ ^*/ \\ & & & r = \hat{T}_e[\![e_2]\!]e\hat{n}v_t \\ & & & \mathbf{p} = (l_{(1)} \rhd r_{(1)}) \sqcup (r_{(1)} \rhd l_{(1)}) \\ & & \text{in} & \langle \mathbf{p}, err \rangle \end{array}$$

Consider an optimized lazy evaluation model in which all strict arguments of a user-defined function are evaluated either sequentially or in parallel before the execution of the function. We assume that each function definition is annotated to indicate in which arguments it is strict. The semantic equation of the abstract before semantic function for $\lambda x_1 . \ldots \lambda x_n . e$ is modified as follows:

$$\begin{array}{rcl} \hat{T}_e[\![\lambda x_1 . \ldots \lambda x_n . e]\!]e\hat{n}v_t & = & \text{let } ap = (\hat{T}_e[\![e]\!]e\hat{n}v_t[x_i \mapsto y_i]) \\ & & \text{in } \langle 1, \lambda y_1 . \langle 1, \ldots \lambda y_n . \langle (1 \rhd \mathbf{s} \rhd ap_{(1)}), ap_{(2)} \rangle \ldots \rangle \rangle \end{array}$$

where

- In sequential (left-to-right) evaluation of strict arguments: $\mathbf{s} = (y_1)_{(1)} \rhd \ldots \rhd (y_n)_{(1)}$ where each $x_i, 1 \leq i \leq n$ is strict parameter.
- In sequential (right-to-left) evaluation of strict arguments: $\mathbf{s} = (y_n)_{(1)} \rhd \ldots \rhd (y_1)_{(1)}$ where each $x_i, 1 \leq i \leq n$ is strict parameter.
- In parallel evaluation of strict arguments: $\mathbf{s} = \bigsqcup_{i \ s.t. \ x_i \ is \ a \ strict \ parameter} (y_i)_{(1)}$

A naive way for the before analysis to handle a language with non-flat domains (due to lists) is that when an object is put in a list we assume that it could be demanded whenever `car` or `cdr` is applied to that list. The abstract before subdomain $\hat{D}_t$ for expressions of type $\tau \ list$ is defined as follows:

$$\hat{D}_t^{\tau \ list} = \hat{D}_t^{\tau} \quad \text{abstract subdomain for lists}$$

The abstract before semantic function $\hat{T}_c$ for constants associated with list-type expressions is defined as follows:

$$
\begin{aligned}
\hat{T}_c[\![\texttt{nil}]\!] &= \langle 1, err \rangle \\
\hat{T}_c[\![\texttt{cons}]\!] &= \langle 1, \lambda x.\langle 1, \lambda y.\langle 2, x_{(2)} \sqcup y_{(2)} \rangle \rangle \rangle \\
\hat{T}_c[\![\texttt{car}]\!] &= \langle 1, \lambda x.x \rangle \\
\hat{T}_c[\![\texttt{cdr}]\!] &= \langle 1, \lambda x.x \rangle \\
\hat{T}_c[\![\texttt{null}]\!] &= \langle 1, \lambda x.\langle x_{(1)}, err \rangle \rangle
\end{aligned}
$$

Clearly, more work remains to be done to provide a more precise analysis for languages with non-flat domains.

Chapter 6

Polymorphic Invariance

All the semantic analyses presented in the preceding chapters have dealt with a higher-order functional language with a *monomorphic* type system in which every expression is rigidly typed. Most modern functional languages adopt a rich polymorphic type system which is more flexible.

In this chapter, we present a method, based on the notion of *polymorphic invariance* ([1], [3]), for applying the escape analysis, the reference escape analysis, and the order-of-demand analysis to a polymorphic language using the analysis techniques for a monomorphic language. First, we describe the notion of polymorphic invariance of information and analysis on polymorphic functional languages. The proofs of polymorphic invariance of the escape analysis, the reference escape analysis, and the order-of-demand analysis are then presented. Finally, we discussed the approach for analyzing polymorphic functions.

6.1 Issues in Analyzing Polymorphic Functions

Most modern functional programming languages support some polymorphism with a flexible polymorphic type system. A function is said to be *polymorphic* if it can be applied uniformly to arguments of a range of types more than one type. We particularly concentrate on a kind of polymorphism, called *parametric (generic) polymorphism*, which most modern functional languages like ML support. Type expressions are parameterized with type parameters and all type parameters are universally quantified at the top level.

Each semantic analysis for monomorphic languages presented in the previous chapters can be applied to polymorphic languages by performing monomorphic semantic analysis to each monomorphic instance of a polymorphic function. There are, however, two problems

connected with this approach ([1], [3]):

- The number of monomorphic instances of a polymorphic function becomes infinite as soon as we allow structured types or higher-order functions.
- The size of the abstract domain for structured and higher-order types goes so fast that fixpoint computations become infeasible

Thus, this approach of performing the semantic analysis for a monomorphic language on each monomorphic instance of a polymorphic function is semantically sound, but is not satisfactory from a pragmatic point of view. Analysis of higher-order polymorphic functions is best done by proving a polymorphic invariance result for the analysis and then computing those non-basic instances of the abstract functions necessary to compute all basic instances of the abstract functions.

6.2 Polymorphic Invariance

We define the notion of the polymorphic invariance of properties and of analyses for polymorphic languages ([1], [3]).

Definition 6.1 (The Polymorphic Invariance of a Property) Let P be a property of expressions over a polymorphic language. P is said to be *polymorphically invariant* if e' satisfies $P \iff e''$ satisfies P for all $e \in Exp$, for all e', e'' in the set of possible monomorphic instances of e.

This says that if a property is polymorphically invariant then for any polymorphic expression e, that property must hold either for all its monomorphic instances or for none.

Definition 6.2 (The Polymorphic Invariance of an Analysis) Let A be an analysis for detecting some property. A is *polymorphically invariant* if the application of the analysis to any two monomorphic instances of a polymorphic function always yields identical results.

This means that if an analysis is polymorphically invariant then a polymorphic function can be analyzed by considering only one of its monomorphic instances, since the result would apply to all its monomorphic instances.

6.3 Polymorphic Invariance Proofs

We think of a polymorphic function as a generic notation for the set of its possible monomorphic instances. When we apply the function in an actual computation, the ultimate effect is that data of a determinate base type is produced, so we can regard each application as one of the monotype instances. This suggest that sets of monomorphic instances for some polymorphically typed expression are handled in a semantically natural fashion, and that some properties should be invariant over such sets.

6.3.1 Escape Analysis

The polymorphic invariance of the escape property of functions implies that whether a parameter escapes from a function or not is independent of the type of the parameter. Thus, given a polymorphic function, it will return the same escape result on any two monomorphic instances of the function. As a consequence of this fact, the escape analysis problem for polymorphic functions can be reduced to the escape analysis problem for monomorphic functions. Since a smaller types means fewer elements of that type, and since the efficiency of escape analysis and similar analyses requiring fixpoint finding is dependent on the number of elements in the domain, the proof that our escape analysis is polymorphically invariant is important. By means of this reduction, it should be possible in practice to confine applications of the escape analysis method to lower-order types.

We prove that our escape analysis is indeed polymorphically invariant, that is, given a polymorphic function, it will return the identical escape result on any two monomorphic instances of the function. We first introduce a relation among all possible monomorphic instances of a polymorphic function which relates them with respect to their escape property. Given a polymorphic expression e, consider any two of its monomorphic instances e' and e'' as follows: e' and e'' are of type τ' and τ'', and n' and n'' are the number of arguments that the types τ' and τ'' can take before returning a primitive value, respectively. Let u' and u'' be the values in the abstract escape domain $\hat{D}_o$ of e' and e'', respectively. We define a notion of *similarity* between u' and u'' with respect to the escape property, written $u' \overset{o}{\sim\!\sim} u''$, which relates the escape property of u' to that of u''. We say that $u' \overset{o}{\sim\!\sim} u''$ iff

$$(\mathrm{NAP}_k(u', s_1, \ldots, s_k))_{(1)} = (\mathrm{NAP}_k(u'', t_1, \ldots, t_k))_{(1)}$$

for all $k \leq n$, where n is the minimum of n' and n'', and for all $i \leq k$, $s_i \overset{o}{\sim\!\sim} t_i$.

Lemma 6.1 *Let f be a polymorphic recursive function defined as $f\ x_1\ \ldots\ x_n\ =\ e$ where e contains free variables $v_1 \ldots v_m$. Let f' and f' be two monomorphic instances of f, typed as follows:*

$$[v_1 : \sigma'_1, \ldots, v_m : \sigma'_m]f' : \tau'_1 \to \ldots \to \tau'_n \to \tau'$$
$$[v_1 : \sigma''_1, \ldots, v_m : \sigma''_m]f'' : \tau''_1 \to \ldots \to \tau''_n \to \tau''$$

where each σ and τ is a monotype. For monotyped abstract escape environments env'_o and env''_o that map each v_i to an element in $\hat{D}_o$ such that for each v_i, $env'_o[\![v_i]\!] \stackrel{o}{\sim} env''_o[\![v_i]\!]$,

$$\hat{f}' \stackrel{o}{\sim} \hat{f}''$$

where $\hat{f}' = \hat{O}_e[\![\lambda x_1 . \ldots \lambda x_n . e]\!] env'_o$ and $\hat{f}'' = \hat{O}_e[\![\lambda x_1 . \ldots \lambda x_n . e]\!] env''_o$.

Proof : Let $e\hat{n}v'_o$ and $e\hat{n}v''_o$ be defined as follows:

$$e\hat{n}v'_o = env'_o[f' \mapsto \hat{O}_e[\![\ f'\ ]\!] e\hat{n}v'_o],$$
$$e\hat{n}v''_o = env''_o[f'' \mapsto \hat{O}_e[\![\ f''\ ]\!] e\hat{n}v''_o]$$

and let $\hat{f}'$ and $\hat{f}''$ be defined as follows:

$$\hat{f}' = e\hat{n}v'_o[\![f']\!]$$
$$\hat{f}'' = e\hat{n}v''_o[\![f'']\!]$$

We can prove $\hat{f}' \stackrel{o}{\sim} \hat{f}''$ by fixpoint induction on $\hat{f}$, i.e. $e\hat{n}v_o$.
(I) Base Case of Fixpoint Induction : The first approximation $\hat{f}'^{(0)}$ of $\hat{f}'$ is $\langle 0, \lambda y_1 . \langle y_{1(1)}, \lambda y_2 . \langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n . \perp_{\tau'} \rangle \ldots \rangle \rangle \rangle$. The first approximation $\hat{f}''^{(0)}$ of $\hat{f}''$ is $\langle 0, \lambda y_1 . \langle y_{1(1)}, \lambda y_2 . \langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n . \perp_{\tau''} \rangle \ldots \rangle \rangle \rangle$. Since $\perp_{\tau'} \stackrel{o}{\sim} \perp_{\tau''}$, it holds.
(II) Fixpoint Induction Step : Assume that $\hat{f'^{(m)}} \stackrel{o}{\sim} \hat{f''^{(m)}}$ for some $m \geq 0$. (fixpoint induction hypothesis) Then, we prove that $\hat{f}'^{(m+1)} \stackrel{o}{\sim} \hat{f''^{(m+1)}}$. This can be proved by structural induction on expression e. Let $e\hat{n}v'^{(m)}_o$ and $e\hat{n}v''^{(m)}_o$ be $env'_o[x_i \mapsto y_i,\ f' \mapsto \hat{f}'^{(m)}]$ and $env''_o[x_i \mapsto y_i],\ f'' \mapsto \hat{f}''^{(m)}]$, respectively.
I. Base Case of Structural Induction:

1. $e = c$: $\hat{f}'^{(m+1)} = \langle 0, \lambda y_1 . \langle y_{1(1)}, \lambda y_2 . \langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n . \hat{O}_c[\![c]\!] \rangle \ldots \rangle \rangle \rangle$. Similarly, $\hat{f}''^{(m+1)} = \langle 0, \lambda y_1 . \langle y_{1(1)}, \lambda y_2 . \langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n . \hat{O}_c[\![c]\!] \rangle \ldots \rangle \rangle \rangle$. Since $\hat{O}_c[\![c]\!] = \hat{O}_c[\![c]\!]$, it holds.

2. $e = x$: $\hat{f}'^{(m+1)} = \langle 0, \lambda y_1 . \langle y_{1(1)}, \lambda y_2 . \langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n . \hat{O}_e[\![x]\!]\ e\hat{n}v'^{(m)}_o \rangle \ldots \rangle \rangle$ and $\hat{f}''^{(m+1)} = \langle 0, \lambda y_1 . \langle y_{1(1)}, \lambda y_2 . \langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n . \hat{O}_e[\![x]\!]\ e\hat{n}v''^{(m)}_o \rangle \ldots \rangle \rangle \rangle$. By the fixpoint induction hypothesis, in either $x = x_i$ or $x = f$, it holds.

II. Structural Induction Step: Assume that $f'^{\hat{}(m)} \stackrel{o}{\sim\sim} f''^{\hat{}(m)}$ for e_1, e_2 and e_3. (structural induction hypothesis)

1. $e = e_1 + e_2$: Since $\hat{f}'^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle 0, err\rangle\rangle \ldots\rangle\rangle$ and $\hat{f}''^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle 0, err\rangle\rangle \ldots\rangle\rangle\rangle$, it holds. Similarly, this holds for $e_1 - e_2$ and $e_1 = e_2$.

2. $e =$ `if` e_1 `then` e_2 `else` e_3 : $\hat{f}'^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{O}_e[\![e_2]\!]\ e\hat{n}v_o'^{(m)} \sqcup \hat{O}_e[\![e_3]\!]\ e\hat{n}v_o'^{(m)})\rangle \ldots\rangle\rangle$. $\hat{f}''^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{O}_e[\![e_2]\!]\ e\hat{n}v_o''^{(m)} \sqcup \hat{O}_e[\![e_3]\!]\ e\hat{n}v_o''^{(m)})\rangle \ldots\rangle\rangle\rangle$. By the fixpoint and structural induction hypotheses, it holds.

3. $e = e_1 e_2$: $\hat{f}'^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{O}_e[\![e_1]\!]\ e\hat{n}v_o'^{(m)})_{(2)}\ (\hat{O}_e[\![e_2]\!]\ e\hat{n}v_o'^{(m)})\rangle \ldots\rangle\rangle\rangle$, and $\hat{f}''^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{O}_e[\![e_1]\!]\ e\hat{n}v_o''^{(m)})_{(2)}\ (\hat{O}_e[\![e_2]\!]\ e\hat{n}v_o''^{(m)})\rangle \ldots\rangle\rangle\rangle$. By the structural induction hypothesis and the definition of $\stackrel{o}{\sim\sim}$, it holds.

4. $e =$ `lambda`$(x).e_1$: $\hat{f}'^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle \hat{V}',\ \lambda y.\hat{O}_e[\![e_1]\!]\ e\hat{n}v_o'^{(m)}[x_i \mapsto y_i]\rangle\ \rangle \ldots\rangle\rangle\rangle$, and $\hat{f}''^{(m+1)} = \langle 0, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle \hat{V}'',\ \lambda y.\hat{O}_e[\![e_1]\!]\ e\hat{n}v_o''^{(m)}[x_i \mapsto y_i]\rangle\ \rangle \ldots\rangle\rangle\rangle$. Since $e\hat{n}v_o'^{(m)}[\![z]\!]\rangle \stackrel{o}{\sim\sim} e\hat{n}v_o''^{(m)}[\![z]\!]\rangle$, $\hat{V}' = \hat{V}''$. Then, by the structural induction hypothesis, it holds.

□

The above lemma says that all possible monomotype instances of a polymorphic function are similar with respect to escape property of their arguments and local objects. Based on this fact, we prove that both the global and local escape analyses are polymorphically invariant.

Theorem 6.1 (Global Escape Analysis) *Let f be a polymorphic function of arity n, and let f' and f'' be any two monomorphic instances of f. Assume that env' and env'' are abstract escape semantic environments that map f' and f'' to elements of $\hat{D}_o$, respectively. Then, for $1 \leq i \leq n$,*

$$\mathsf{G_escape?}(f', i, env') = \mathsf{G_escape?}(f'', i, env'')$$

Proof : From the definition of the global escape test function,

$$\mathsf{G_escape?}(f', i, env') = (\hat{O}_e[\![f\ x_1\ \ldots\ x_n]\!]\ env'[x_i \mapsto y_i])_{(1)}$$

where $y_i' = \langle 1, W^{\tau_i'}\rangle$ and, for $j \leq n$ and $j \neq i$, $y_j' = \langle 0, W^{\tau_j'}\rangle$. Let $\hat{f}' = env'[\![f]\!]$. Then, by the definition of $\hat{O}_e$,

$$\mathsf{G_escape?}(f', i, env') = (\mathrm{NAP}_n(\hat{f}', y'_1, \ldots, y'_n))_{(1)}.$$

Similarly,

$$\mathsf{G_escape?}(f'', i, env') = (\mathrm{NAP}_n(\hat{f}'', y''_1, \ldots, y''_n))_{(1)}$$

where $\hat{f}'' = env''[\![f]\!]$, $y''_i = \langle 1, W^{\tau''_i} \rangle$ and, for $j \leq n$ and $j \neq i$, $y''_j = \langle 0, W^{\tau''_j} \rangle$. By the definition of the worst-case escape function W, $y'_k \overset{\circ}{\sim}\sim y''_k$ for all $1 \leq k \leq n$. Thus, by the Lemma 6.1, $\hat{f}' \overset{\circ}{\sim}\sim \hat{f}''$. Then, by definition of $\overset{\circ}{\sim}\sim$, we have that $(\mathrm{NAP}_n(\hat{f}', y'_1, \ldots, y'_n))_{(1)}$ $= (\mathrm{NAP}_n(\hat{f}'', y''_1, \ldots, y''_n))_{(1)}$. Thus, we conclude that

$$\mathsf{G_escape?}(f', i, env') = \mathsf{G_escape?}(f'', i, env'').$$

□

Theorem 6.2 (Local Escape Analysis) *Let f be a polymorphic function of arity n in an application $f\ e_1\ \ldots\ e_n$. Let f' and f'' be any two monomorphic instances of f, and e'_i and e''_i be two monomorphic instances of e_i. Assume that env' and env'' are abstract escape semantic environments that map f' and all free identifiers within e'_i, and f'' and all free identifiers within e''_i to elements of $\hat{D}_o$, respectively. Then, for $1 \leq i \leq n$,*

$$\mathsf{L_escape?}(f', i, e'_1, \ldots, e'_n, env') = \mathsf{L_escape?}(f'', i, e''_1, \ldots, e''_n, env'')$$

Proof : This can be proved in a similar way to the polymorphic invariance proof of the global escape analysis. □

6.3.2 Refined Escape Analysis

The polymorphic invariance of the refined escape property of functions implies that the extent of a parameter which does not escape from the function call is independent of the type of the parameter. Thus, given a polymorphic function, it will return the same refined escape result on any two monomorphic instances of the function. The polymorphic invariance of the refined escape analysis means that given a polymorphic function, it will return the same refined escape result for any two monomorphic instances of that function. Actually, the refined escape analysis is polymorphically invariant when it is stated the following way:

> Given a polymorphic function, the number of spines of a parameter that does not escape from the function application is the same for any two monomorphic instances of the function.

From this point of view, we will prove that our refined escape analysis is indeed polymorphically invariant. We introduce a relation among all possible monomorphic instances of a polymorphic function which relates them with respect to their refined escape property as stated above. Given a polymorphic expression e, consider any two its monomorphic instances e' and e'' as follows: e' and e'' are of type τ' and τ'', and n' and n'' are the number of arguments that the types τ' and τ'' can take before returning a primitive value, respectively. Let u' and u'' be the values in the abstract refined escape domain $\hat{D}_p$ of e' and e'', respectively. We define a notion of *similarity* between u' and u'' with respect to the refined escape property, written $u' \overset{p}{\sim}\sim u''$, which relates the refined escape property of u' to that of u''. We say that $u' \overset{p}{\sim}\sim u''$ iff

$$\mathrm{NAP}_k(u', s_1, \ldots, s_k))_{(1)} = \mathrm{NAP}_k(u'', t_1, \ldots, t_k))_{(1)} = \langle 0, 0 \rangle$$

or

$$\mathrm{NAP}_n(u', s_1, \ldots, s_k))_{(1)(1)} = \mathrm{NAP}_n(u'', t_1, \ldots, t_k))_{(1)(1)} = 1 \text{ and}$$
$$d' - \mathrm{NAP}_n(u', s_1, \ldots, s_k))_{(1)(2)} = d'' - \mathrm{NAP}_n(u'', t_1, \ldots, t_k))_{(1)(2)}$$

for all $k \leq n$ where n is the minimum of n' and n'', for all $j \leq k$, $(s_j)_{(1)} = \langle 1, d'_j \rangle$ and $(t_j)_{(1)} = \langle 1, d''_j \rangle$ where d'_j and d''_j is the number of spines of the j^{th} parameter of that n parameters, for all $l \neq j$ and $l \leq k$, $(s_l)_{(1)} = (t_l)_{(1)} = \langle 0, 0 \rangle$, and for all $i \leq k$, $s_i \overset{p}{\sim}\sim t_i$.

Lemma 6.2 *Let f be a polymorphic recursive function defined as $f\ x_1\ \ldots\ x_n\ =\ e$ where e contains free variables $v_1 \ldots v_m$. Let f' and f' be two monomorphic instances of f, typed as follows:*

$$[v_1 : \sigma'_1, \ldots, v_m : \sigma'_m]f' : \tau'_1 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \tau'$$
$$[v_1 : \sigma''_1, \ldots, v_m : \sigma''_m]f'' : \tau''_1 \rightarrow \ldots \rightarrow \tau''_n \rightarrow \tau''$$

where each σ and τ is a monotype. For monotyped abstract refined escape environments env'_p and env''_p that map each v_i to an element in $\hat{D}_p$ such that for each v_i, $env'_p[\![v_i]\!] \overset{p}{\sim}\sim env''_p[\![v_i]\!]$,

$$\hat{f}' \overset{p}{\sim}\sim \hat{f}''$$

where $\hat{f}' = \hat{P}_e[\![\lambda x_1 \ldots . \lambda x_n . e]\!] env'_p$ and $\hat{f}'' = \hat{P}_e[\![\lambda x_1 \ldots . \lambda x_n . e]\!] env''_p$.

Proof : Let $e\hat{n}v'_p$ and $e\hat{n}v''_p$ be defined as follows:

$$e\hat{n}v'_p = env'_p[f' \mapsto \hat{P}_e[\![\ f'\ ]\!] e\hat{n}v'_p],$$
$$e\hat{n}v''_p = env''_p[f'' \mapsto \hat{P}_e[\![\ f''\ ]\!] e\hat{n}v''_p]$$

and let $\hat{f'}$ and $\hat{f''}$ be defined as follows:

$$\hat{f'} = e\hat{n}v'_p[\![f']\!]$$
$$\hat{f''} = e\hat{n}v''_p[\![f'']\!]$$

We can prove this by fixpoint induction on $\hat{f}$, i.e. $e\hat{n}v_p$.
(I) Base Case of Fixpoint Induction : The first approximation $\hat{f'}^{(0)}$ of $\hat{f'}$ $\langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\perp_{\tau'}\rangle \ldots\rangle\rangle\rangle$. The first approximation $\hat{f''}^{(0)}$ of $\hat{f''}$ $\langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\perp_{\tau''}\rangle \ldots\rangle\rangle\rangle$. Since $\perp_{\tau'} \overset{p}{\sim}\sim \perp_{\tau'}$, it holds.
(II) Fixpoint Induction Step : Assume that $\hat{f'}^{(m)} \overset{p}{\sim}\sim \hat{f''}^{(m)}$ for some $m \geq 0$. (fixpoint induction hypothesis) Then, we prove that $\hat{f'}^{(m+1)} \overset{p}{\sim}\sim \hat{f''}^{(m+1)}$. This can be proved by structural induction on expression e. Let $e\hat{n}v_p^{'(m)}$ and $e\hat{n}v_p^{''(m)}$ be $env'_p[x_i \mapsto y_i,\ f' \mapsto \hat{f'}^{(m)}]$ and $env''_p[x_i \mapsto y_i],\ f'' \mapsto \hat{f''}^{(m)}]$, respectively.
I. Base Case of Structural Induction:

1. $e = c$: $\hat{f'}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\ \langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{P}_c[\![c]\!]\rangle \ldots\rangle\rangle\rangle$. Similarly, $\hat{f''}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{P}_c[\![c]\!]\rangle \ldots\rangle\rangle\rangle$. Thus, clearly it holds.

2. $e = x$: $\hat{f'}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{P}_e[\![x]\!]\ e\hat{n}v_p^{'(m)}\rangle \ldots\rangle\rangle$ and $\hat{f''}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{P}_e[\![x]\!]\ e\hat{n}v_p^{''(m)}\rangle \ldots\rangle\rangle\rangle$. By the fixpoint induction hypothesis, in either $x = x_i$ or $x = f$, it holds.

II. Structural Induction Step: Assume that $\hat{f'}^{(m)} \overset{p}{\sim}\sim \hat{f''}^{(m)}$ for e_1, e_2 and e_3. (structural induction hypothesis)

1. $e = e_1 + e_2$: Since $\hat{f'}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle\langle 0,0\rangle, err\rangle\rangle \ldots\rangle\rangle$ and $\hat{f''}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle\langle 0,0\rangle, err\rangle\rangle \ldots\rangle\rangle\rangle$, it holds. Similarly, this holds for $e_1 - e_2$ and $e_1 = e_2$.

2. $e =$ `if` e_1 `then` e_2 `else` e_3 : $\hat{f'}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{P}_e[\![e_2]\!]\ e\hat{n}v_p^{'(m)} \sqcup \hat{P}_e[\![e_3]\!]\ e\hat{n}v_p^{'(m)})\rangle \ldots\rangle\rangle$ and $\hat{f''}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{P}_e[\![e_2]\!]\ e\hat{n}v_p^{''(m)} \sqcup \hat{P}_e[\![e_3]\!]\ e\hat{n}v_p^{''(m)})\rangle \ldots\rangle\rangle\rangle$. By the fixpoint and structural induction hypotheses, it holds.

3. $e = e_1 e_2$: $\hat{f'}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.(\hat{P}_e[\![e_1]\!]\ e\hat{n}v_p^{'(m)})_{(2)}\ (\hat{P}_e[\![e_2]\!]\ e\hat{n}v_p^{'(m)})\rangle \ldots\rangle\rangle\rangle$. $\hat{f''}^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.(\hat{P}_e[\![e_1]\!]\ e\hat{n}v_p^{''(m)})_{(2)}\ (\hat{P}_e[\![e_2]\!]\ e\hat{n}v_p^{''(m)})\rangle \ldots\rangle\rangle\rangle$. By the structural induction hypothesis and the definition of $\overset{p}{\sim}\sim$, it holds.

4. $e = \texttt{lambda}(x).e_1 : \hat{f}'^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle \hat{V}', \lambda y.\hat{P}_e[\![e_1]\!]$ $e\hat{n}v_p^{'(m)}[x_i \mapsto y_i]\rangle \rangle \ldots \rangle\rangle\rangle$, and $\hat{f}''^{(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\langle \hat{V}'',$ $\lambda y.\hat{P}_e[\![e_1]\!]\ e\hat{n}v_p^{''(m)}[x_i \mapsto y_i]\rangle \rangle \ldots \rangle\rangle\rangle$. Since $e\hat{n}v_p^{'(m)}[\![z]\!]\rangle \overset{p}{\sim}\sim e\hat{n}v_p^{''(m)}[\![z]\!]\rangle$, $\hat{V}' = \hat{V}''$. Then, by the structural induction hypothesis, it holds.

□

The above lemma says that all possible monomorphic instances of a polymorphic function are related with respect to refined escape property of their arguments and local objects. Based on this fact, we prove that both the global and local refined escape analyses are polymorphically invariant.

Theorem 6.3 (Global Refined Escape Analysis) *Let f be a polymorphic function of arity n, and let f' and f'' be any two monomorphic instances of f. Assume that env' and env'' are abstract refined escape semantic environments that map f' and f'' to elements of $\hat{D}_p$, respectively. Then, for $1 \leq i \leq n$,*

$$\mathsf{G_rescape?}(f', i, env') = \langle 0,0\rangle \Longleftrightarrow \mathsf{G_rescape?}(f'', i, env'') = \langle 0,0\rangle$$

or

$$\mathsf{G_rescape?}(f', i, env') = \langle 1,k'\rangle \Longleftrightarrow \mathsf{G_rescape?}(f'', i, env'') = \langle 1,k''\rangle$$
such that $s_i' - k' = s_i'' - k''$ where s_i' and s_i'' are the number of spines of the i^{th} parameter of f' and f'', respectively.

Proof : From the definition of the global refined escape test function,

$$\mathsf{G_rescape?}(f', i, env') = (\hat{P}_e[\![f\ x_1\ \ldots\ x_n]\!]\ env'[x_i \mapsto y_i])_{(1)}$$

where $y_i' = \langle\langle 1, s_i'\rangle, W^{\tau_i'}\rangle$ and, for $j \leq n$ and $j \neq i$, $y_j' = \langle\langle 0,0\rangle W^{\tau_j'}\rangle$. Let $\hat{f}' = env'[\![f]\!]$. Then, by the definition of $\hat{P}_e$, Then,

$$\mathsf{G_rescape?}(f', i, env') = (\mathrm{NAP}_n(\hat{f}', y_1', \ldots, y_n'))_{(1)}.$$

Similarly,

$$\mathsf{G_rescape?}(f'', i, env') = (\mathrm{NAP}_n(\hat{f}'', y_1'', \ldots, y_n''))_{(1)}$$

where $\hat{f}'' = env''[\![f]\!]$, $y_i'' = \langle\langle 1, s_i''\rangle, W^{\tau_i''}\rangle$ and, for $j \leq n$ and $j \neq i$, $y_j'' = \langle\langle 0,0\rangle W^{\tau_j''}\rangle$. By the definition of the worst-case escape function W, $y_k' \overset{p}{\sim}\sim y_k''$ for all $1 \leq k \leq n$. Thus, by the Lemma 6.2, $\hat{f}' \overset{p}{\sim}\sim \hat{f}''$. Then, by definition of $\overset{p}{\sim}\sim$, we have that

$$\mathrm{NAP}_n(\hat{f}', y_1', \ldots, y_n') = \mathrm{NAP}_n(\hat{f}'', y_1'', \ldots, y_n'') = \langle 0,0\rangle$$

or

$$(\mathrm{NAP}_n(\hat{f}', y_1', \ldots, y_n'))_{(1)(1)} = (\mathrm{NAP}_n(\hat{f}'', y_1'', \ldots, y_n''))_{(1)(2)} = 1$$

and

$$s_i' - (\mathrm{NAP}_n(\hat{f}', y_1', \ldots, y_n'))_{(1)(2)} = s_i'' - (\mathrm{NAP}_n(\hat{f}'', y_1'', \ldots, y_n''))_{(1)(2)}$$

Thus, we conclude that it holds. □

Theorem 6.4 (Local Refined Escape Analysis) *Let f be a polymorphic function of arity n in an application $f\ e_1\ \ldots\ e_n$. Let f' and f'' be any two monomorphic instances of f, and e_i' and e_i'' be two monomorphic instances of e_i. Assume that env' and env'' are abstract escape semantic environments that map f' and all free identifiers within e_i', and f'' and all free identifiers within e_i'' to elements of $\hat{D}_p$, respectively. Then, for $1 \leq i \leq n$,*

$$\mathsf{L_rescape?}(f', i, e_1', \ldots, e_n', env') = \langle 0, 0 \rangle$$
$$\Longleftrightarrow \mathsf{L_rescape?}(f'', i, e_1'', \ldots, e_n'', env'') = \langle 0, 0 \rangle$$

or

$$\mathsf{L_rescape?}(f', i, e_1', \ldots, e_n', env') = \langle 0, 0 \rangle = \langle 1, k' \rangle$$
$$\Longleftrightarrow \mathsf{L_rescape?}(f'', i, e_1'', \ldots, e_n'', env'') = \langle 0, 0 \rangle = \langle 1, k'' \rangle$$

such that $s_i' - k' = s_i'' - k''$ where s_i' and s_i'' are the number of spines of e_i' and e_i'', respectively.

Proof : This can be proved in a similar way to the polymorphic invariance proof of the global refined escape analysis. □

6.3.3 Reference Escape Analysis

The polymorphic invariance of the reference escape property of functions says that whether a reference to a heap allocated object escapes a call to a polymorphic function or not is independent of the type of the object to which the reference is pointing. This means that given a polymorphic function, reference escape analysis will provide the same reference escape result on any two monomorphic instances of that function.

We prove that our reference escape analysis is indeed polymorphically invariant. We introduce a relation among all possible monomorphic instances of a polymorphic function which relates them with respect to their reference escape property. Given a polymorphic expression e, consider any two its monomorphic instances e' and e'' as follows: e' and e'' are of type τ' and τ'', and n' and n'' are the number of arguments that the types τ' and τ''

can take before returning a primitive value, respectively. Let u' and u'' be the values in the abstract reference escape domain $\hat{D}_r$ of e' and e'', respectively. Let u' and u'' be the values in the abstract reference escape domain $\hat{D}_r$ of e' and e'', respectively. We define a notion of *similarity* between u' and u'' with respect to reference escape property, written $u' \stackrel{r}{\sim\!\sim} u''$, which relates the reference escape property of u' to that of u''. We say that $u' \stackrel{r}{\sim\!\sim} u''$ iff

$$(\mathrm{NAP}_k(u', s_1, \ldots, s_k))_{(1)(1)} = (\mathrm{NAP}_k(u'', t_1, \ldots, t_k))_{(1)(1)}$$

for all $k \leq n$ where n is the minimum of n' and n'', and for all $i \leq k$, $s_i \stackrel{r}{\sim\!\sim} t_i$.

Lemma 6.3 *Let f be a polymorphic recursive function defined as $f\ x_1\ \ldots\ x_n\ =\ e$ where e contains free variables $v_1 \ldots v_m$. Let f' and f' be two monomorphic instances of f, typed as follows:*

$$[v_1 : \sigma'_1, \ldots, v_m : \sigma'_m]f' : \tau'_1 \to \ldots \to \tau'_n \to \tau'$$
$$[v_1 : \sigma''_1, \ldots, v_m : \sigma''_m]f'' : \tau''_1 \to \ldots \to \tau''_n \to \tau''$$

where each σ and τ is a monotype. For monotyped abstract reference escape environments env'_r and env''_r that map each v_i to an element in $\hat{D}_r$ and for each v_i, $env'_r[\![v_i]\!] \stackrel{r}{\sim\!\sim} env''_r[\![v_i]\!]$, respectively,

$$\hat{f}' \stackrel{r}{\sim\!\sim} \hat{f}''$$

where $\hat{f}' = \hat{R}_e[\![\lambda x_1 \ldots . \lambda x_n . e]\!]env'_r$ and $\hat{f}'' = \hat{R}_e[\![\lambda x_1 \ldots . \lambda x_n . e]\!]env''_r$.

Proof : Let $e\hat{n}v'_r$ and $e\hat{n}v''_r$ be defined as follows:

$$e\hat{n}v'_r = env'_r[f' \mapsto \hat{R}_e[\![\ f'\ ]\!]e\hat{n}v'_r],$$
$$e\hat{n}v''_r = env''_r[f'' \mapsto \hat{R}_e[\![\ f''\ ]\!]e\hat{n}v''_r]$$

and let $\hat{f}'$ and $\hat{f}''$ be defined as follows:

$$\hat{f}' = e\hat{n}v'_r[\![f']\!]$$
$$\hat{f}'' = e\hat{n}v''_r[\![f'']\!]$$

We can prove $\hat{f}' \stackrel{r}{\sim\!\sim} \hat{f}''$ by fixpoint induction on $\hat{f}$, i.e. $e\hat{n}v_r$.
(I) Base Case of Fixpoint Induction : The first approximation $\hat{f'}^{(0)}$ of $\hat{f}'$ is $\langle\langle 0, 0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\perp_{\tau'}\rangle \ldots .\rangle\rangle\rangle$. The first approximation $\hat{f''}^{(0)}$ of $\hat{f}''$ is $\langle\langle 0, 0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots \lambda y_n.\perp_{\tau''}\rangle \ldots .\rangle\rangle\rangle$. Since $\perp_{\tau'} \stackrel{r}{\sim\!\sim} \perp_{\tau'}$, it holds.
(II) Fixpoint Induction Step : Assume that $\hat{f'}^{(m)} \stackrel{r}{\sim\!\sim} \hat{f''}^{(m)}$ for some $m \geq 0$. (fixpoint induction hypothesis) Then, we prove that $\hat{f'}^{(m+1)} \stackrel{r}{\sim\!\sim} \hat{f''}^{(\hat{m}+1)}$. This can be proved by

structural induction on expression e. Let $e\hat{n}v_r^{'(m)}$ and $e\hat{n}v_r^{''(m)}$ be $env_r'[x_i \mapsto y_i,\ f' \mapsto \hat{f}^{'(m)}]$ and $env_r''[x_i \mapsto y_i],\ f'' \mapsto \hat{f}^{''(m)}]$, respectively.

I. Base Case of Structural Induction:

1. $e = c$: $\hat{f}^{'(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\ \langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{R}_c[\![c]\!]\rangle \ldots\rangle\rangle\rangle$. Similarly, $\hat{f}^{''(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{R}_c[\![c]\!]\rangle \ldots\rangle\rangle\rangle$. Thus, clearly it holds.

2. $e = x$: $\hat{f}^{'(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{R}_e[\![x]\!]\ e\hat{n}v_r^{'(m)}\rangle \ldots\rangle\rangle$ and $\hat{f}^{''(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\hat{R}_e[\![x]\!]\ e\hat{n}v_r^{''(m)}\rangle \ldots\rangle\rangle\rangle$. By the fixpoint induction hypothesis, in either $x = x_i$ or $x = f$, it holds.

II. Structural Induction Step: Assume that $f'\hat{}^{(m)} \sim^r\!\sim f''\hat{}^{(m)}$ for e_1, e_2 and e_3. (structural induction hypothesis)

1. $e = e_1 + e_2$: Since $\hat{f}^{'(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\langle\langle 0,0\rangle, err\rangle\rangle \ldots\rangle\rangle$ and $\hat{f}^{''(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\langle\langle 0,0\rangle, err\rangle\rangle \ldots\rangle\rangle\rangle$, it holds. Similarly, this holds for $e_1 - e_2$ and $e_1 = e_2$.

2. $e =$ `if` e_1 `then` e_2 `else` e_3 : $\hat{f}^{'(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.(\hat{R}_e[\![e_2]\!]\ e\hat{n}v_r^{'(m)} \sqcup \hat{R}_e[\![e_3]\!]\ e\hat{n}v_r^{'(m)})\rangle \ldots\rangle\rangle$ and $\hat{f}^{''(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.(\hat{R}_e[\![e_2]\!]\ e\hat{n}v_r^{''(m)} \sqcup \hat{R}_e[\![e_3]\!]\ e\hat{n}v_r^{''(m)})\rangle \ldots\rangle\rangle\rangle$. By the fixpoint and structural induction hypotheses, it holds.

3. $e = e_1 e_2$: $\hat{f}^{'(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.(\hat{R}_e[\![e_1]\!]\ e\hat{n}v_r^{'(m)})_{(2)}\ (\hat{R}_e[\![e_2]\!]\ e\hat{n}v_r^{'(m)})\rangle \ldots\rangle\rangle\rangle$. $\hat{f}^{''(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.(\hat{R}_e[\![e_1]\!]\ e\hat{n}v_r^{''(m)})_{(2)}\ (\hat{R}_e[\![e_2]\!]\ e\hat{n}v_r^{''(m)})\rangle \ldots\rangle\rangle\rangle$. By the structural induction hypothesis and the definition of $\sim^r\!\sim$, it holds.

4. $e =$ `lambda`$(x).e_1$: $\hat{f}^{'(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\langle \hat{V}', \lambda y.\hat{R}_e[\![e_1]\!]\ e\hat{n}v_r^{'(m)}[x_i \mapsto y_i]\rangle\ \rangle \ldots\rangle\rangle\rangle$, and $\hat{f}^{''(m+1)} = \langle\langle 0,0\rangle, \lambda y_1.\langle y_{1(1)}, \lambda y_2.\langle y_{1(1)} \sqcup y_{2(1)}, \ldots\ \lambda y_n.\langle \hat{V}'', \lambda y.\hat{R}_e[\![e_1]\!]\ e\hat{n}v_r^{''(m)}[x_i \mapsto y_i]\rangle\ \rangle \ldots\rangle\rangle\rangle$. Since $e\hat{n}v_r^{'(m)}[\![z]\!]\rangle \sim^r\!\sim e\hat{n}v_r^{''(m)}[\![z]\!]\rangle$, $\hat{V}' = \hat{V}''$. Then, by the structural induction hypothesis, it holds.

□

The above lemma says that all possible monomorphic instances of a polymorphic function are similar with respect to reference escape property of references created within them. Based on this fact, we prove that both the global and local reference escape analyses are polymorphically invariant.

Theorem 6.5 (Global Reference Escape Analysis) *Let f be a polymorphic function of arity n, and let f' and f'' be any two monomorphic instances of f. Assume that env' and env'' are abstract reference escape semantic environments that map f' and f'' to elements of $\hat{D}_r$, respectively. Then, for $1 \leq i \leq n$ and for $1 \leq j \leq o(i)$,*

$$\mathsf{G_refescape?}(f', i, j, env') = \mathsf{G_refescape?}(f'', i, j, env'').$$

Proof : Let f'_a and f''_a be the auxiliary functions for f' and f'', respectively. From the definition of the global reference escape test,

$$\mathsf{G_refescape?}(f', i, j, env') = (\hat{R}_e[\![f'_a\ x_{11}\ \ldots\ x_{no(n)}]\!]env'[f'_a \mapsto \hat{f'}_a, x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where $\hat{f'} = \hat{R}_e[\![f']\!]env'$, $\hat{f}'_a = \hat{R}_e[\![f'_a]\!]env'[f' \mapsto \hat{f'}]$, $y'_{ij} = \langle\langle 1, s'_i\rangle, W^{\tau'_i}\rangle$, for all $k \leq o(i)$ and $k \neq j$, $y'_{ik} = \langle\langle 0, 0\rangle, W^{\tau'_i}\rangle$, and for all $1 \leq m \leq o(l)$ and $l \neq i$, $y'_{lm} = \langle\langle 0, 0\rangle, W^{\tau'_l}\rangle$. Then, by the definition of $\hat{R}_e$,

$$\mathsf{G_refescape?}(f', i, j, env') = (\mathrm{NAP}_m(\hat{f}'_a, y'_{11}, \ldots, y'_{no(n)}))_{(1)(1)}$$

where $m = \sum_i o(i)$. Similarly,

$$\mathsf{G_refescape?}(f'', i, j, env'') = (\mathrm{NAP}_m(\hat{f}''_a, y''_{11}, \ldots, y''_{no(n)}))_{(1)(1)}$$

where $\hat{f''} = \hat{R}_e[\![f'']\!]env''$, $\hat{f}''_a = \hat{R}_e[\![f''_a]\!]env''[f'' \mapsto \hat{f''}]$, $y''_{ij} = \langle\langle 1, s''_i\rangle, W^{\tau''_i}\rangle$, for all $k \leq o(i)$ and $k \neq j$, $y''_{ik} = \langle\langle 0, 0\rangle, W^{\tau''_i}\rangle$, and for all $1 \leq m \leq o(l)$ and $l \neq i$, $y''_{lm} = \langle\langle 0, 0\rangle, W^{\tau''_l}\rangle$. By Lemma 6.3, $\hat{f'} \overset{r}{\sim\sim} \hat{f''}$ and thus $\hat{f}'_a \overset{r}{\sim\sim} \hat{f}''_a$. By the definitions of the worst-case escape function W, $y'_{ij} \overset{r}{\sim\sim} y''_{ij}$ for all $1 \leq i \leq n$ and $1 \leq j \leq o(i)$. Then, by the definition of $\overset{r}{\sim\sim}$,

$$\mathrm{NAP}_m(\hat{f}'_a, y'_{11}, \ldots, y'_{no(n)})_{(1)} = \mathrm{NAP}_m(\hat{f}''_a, y''_{11}, \ldots, y''_{no(n)})_{(1)}.$$

Thus, we conclude that

$$\mathsf{G_refescape?}(f', i, j, env') = \mathsf{G_refescape?}(f'', i, j, env'')$$

□

Theorem 6.6 (Local Reference Escape Analysis) *Let f be a polymorphic function of arity n in an application $f\ e_1\ \ldots\ e_n$. Let f' and f'' be any two monomorphic instances of f, and e'_i and e''_i be two monomorphic instances of e_i. Assume that env' and env'' are abstract escape semantic environments that map f' and all free identifiers within e'_i, and f'' and all free identifiers within e''_i to elements of $\hat{D}_r$, respectively. Then, for $1 \leq i \leq n$ and for $1 \leq j \leq o(i)$,*

$$\mathsf{L_refescape?}(f', i, j, e'_1, \ldots, e'_n, env') = \mathsf{L_refescape?}(f'', i, j, e''_1, \ldots, e''_n, env'')$$

Proof : This can be proved in a similar way to the polymorphic invariance proof of the global reference escape analysis. □

As a consequence of this fact, the reference escape analysis problem for polymorphic functions can be reduced to the problem for monomorphic functions. The reference escape analysis algorithm need only be applied to the simplest monomorphic instance of a function. Smaller types implies fewer elements of that type, and the efficiency of reference escape analysis and similar analyses requiring fixpoint finding is dependent on the number of elements in the domain.

6.3.4 Order-of-Demand Analysis

The polymorphic invariance of the order-of-demand property of functions implies that the order of demand between parameters of a function during the evaluation of the function is independent of the type of the parameters. Thus, given a polymorphic function, it will return the same escape result on any two monomorphic instances of the function.

We prove that our before analysis is indeed polymorphically invariant. We introduce a relation among all possible monomorphic instances of a polymorphic function which relates them with respect to their order-of-demand property. Given a polymorphic expression e, consider any two its monomorphic instances e' and e'' as follows: e' and e'' are of type τ' and τ'', and n' and n'' are the number of arguments that the types τ' and τ'' can take before returning a primitive value, respectively. Let u' and u'' be the values in the abstract before domain $\hat{D}_t$ of e' and e'', respectively. We define a notion of *similarity* between u' and u'' with respect to order-of-demand property, written $u' \overset{t}{\sim}\sim u''$, which relates the before property of u' to that of u''. We say that $u' \overset{t}{\sim}\sim u''$ iff

$$(\mathrm{NAP}_k(u', s_1, \ldots, s_k))_{(1)} = (\mathrm{NAP}_k(u'', t_1, \ldots, t_k))_{(1)}$$

for all $k \leq n$ where n is the minimum of n' and n'', and for all $i \leq k$, $s_i \overset{t}{\sim}\sim t_i$.

Lemma 6.4 *Let f be a polymorphic recursive function defined as $f\ x_1\ \ldots\ x_n = e$ where e contains free variables $v_1 \ldots v_m$. Let f' and f' be two monomorphic instances of f, typed as follows:*

$$[v_1 : \sigma'_1, \ldots, v_m : \sigma'_m]f' : \tau'_1 \to \ldots \to \tau'_n \to \tau'$$
$$[v_1 : \sigma''_1, \ldots, v_m : \sigma''_m]f'' : \tau''_1 \to \ldots \to \tau''_n \to \tau''$$

where each σ and τ is a monotype. For monotyped abstract before environments env'_t and env''_t that map each v_i to an element in $\hat{D}_t$ and for each v_i, $env'_t[\![v_i]\!] \stackrel{t}{\sim\sim} env''_t[\![v_i]\!]$, respectively,

$$\hat{f}' \stackrel{t}{\sim\sim} \hat{f}''$$

where $\hat{f}' = \hat{T}_e[\![\lambda x_1 \ldots \lambda x_n.e]\!]env'_t$ and $\hat{f}'' = \hat{T}_e[\![\lambda x_1 \ldots \lambda x_n.e]\!]env''_t$.

Proof : Let $e\hat{n}v'_t$ and $e\hat{n}v''_t$ be defined as follows:

$$e\hat{n}v'_t = env'_t[f' \mapsto \hat{T}_e[\![\ f'\ ]\!]e\hat{n}v'_t],$$
$$e\hat{n}v''_t = env''_t[f'' \mapsto \hat{T}_e[\![\ f''\ ]\!]e\hat{n}v''_t]$$

and let $\hat{f}'$ and $\hat{f}''$ be defined as follows:

$$\hat{f}' = e\hat{n}v'_t[\![f']\!]$$
$$\hat{f}'' = e\hat{n}v''_t[\![f'']\!]$$

We can prove $\hat{f}' \stackrel{t}{\sim\sim} \hat{f}''$ by fixpoint induction on $\hat{f}$, i.e. $e\hat{n}v_t$.
(I) Base Case of Fixpoint Induction : The first approximation $\hat{f}'^{(0)}$ of $\hat{f}'$ is $\langle 1, \lambda y_1.\langle 1, \lambda y_2.\ \langle 1, \ldots \lambda y_n.\perp_{\tau'}\rangle \ldots\rangle\rangle\rangle$. The first approximation $\hat{f}''^{(0)}$ of $\hat{f}''$ is $\langle 1, \lambda y_1.\langle 1, \lambda y_2.\ \langle 1, \ldots \lambda y_n.\perp_{\tau''}\rangle \ldots\rangle\rangle\rangle$. Since $\perp_{\tau'} \stackrel{t}{\sim\sim} \perp_{\tau''}$, it holds.
(II) Fixpoint Induction Step : Assume that $\hat{f'^{(m)}} \stackrel{t}{\sim\sim} \hat{f''^{(m)}}$ for some $m \geq 0$. (fixpoint induction hypothesis) Then, we prove that $\hat{f}'^{(m+1)} \stackrel{t}{\sim\sim} \hat{f''^{(m+1)}}$. This can be proved by structural induction on expression e. Let $e\hat{n}v_t^{'(m)}$ and $e\hat{n}v_t^{''(m)}$ be $env'_t[x_i \mapsto y_i,\ f' \mapsto \hat{f}'^{(m)}]$ and $env''_t[x_i \mapsto y_i],\ f'' \mapsto \hat{f}''^{(m)}]$, respectively.
I. Base Case of Structural Induction:

1. $e = c$: $\hat{f}'^{(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\ \langle 1, \ldots\ \lambda y_n.\hat{T}_c[\![c]\!]\rangle \ldots\rangle\rangle\rangle$. Similarly, $\hat{f}''^{(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots\ \lambda y_n.\hat{T}_c[\![c]\!]\rangle \ldots\rangle\rangle\rangle$. Thus, clearly it holds.
2. $e = x$: $\hat{f}'^{(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots\ \lambda y_n.\hat{T}_e[\![x]\!]\ e\hat{n}v_t^{'(m)}\rangle \ldots\rangle\rangle\rangle$. $\hat{f}''^{(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots\ \lambda y_n.\hat{T}_e[\![x]\!]\ e\hat{n}v_t^{''(m)}\rangle \ldots\rangle\rangle\rangle$. By the fixpoint induction hypothesis, in either $x = x_i$ or $x = f$, it holds.

II. Structural Induction Step: Assume that $\hat{f'^{(m)}} \stackrel{t}{\sim\sim} \hat{f''^{(m)}}$ for e_1, e_2 and e_3. (structural induction hypothesis)

1. $e = e_1 + e_2$: Let $l' = \hat{T}_e[\![e_1]\!]e\hat{n}v_t^{'(m)}$ and $r' = \hat{T}_e[\![e_2]\!]e\hat{n}v_t^{'(m)}$. Let $l'' = \hat{T}_e[\![e_1]\!]e\hat{n}v_t^{''(m)}$ and $r'' = \hat{T}_e[\![e_2]\!]e\hat{n}v_t^{''(m)}$. Then, $\hat{f}'^{(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots\ \lambda y_n.\langle l'_{(1)} \rhd r'_{(1)}, err\rangle\rangle \ldots\rangle\rangle$ and $\hat{f}''^{(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots\ \lambda y_n.\langle l''_{(1)} \rhd r''_{(1)}, err\rangle\rangle \ldots\rangle\rangle\rangle$. By the structural induction hypothesis and the definition of $\stackrel{t}{\sim\sim}$, it holds.

2. e = `if` e_1 `then` e_2 `else` e_3 : Let $p' = \hat{T}_e[\![e_1]\!]e\hat{n}v_t^{'(m)}$, $c' = \hat{T}_e[\![e_2]\!]e\hat{n}v_t^{'(m)}$, and $a' = \hat{T}_e[\![e_3]\!]e\hat{n}v_t^{'(m)}$. Let $p'' = \hat{T}_e[\![e_1]\!]e\hat{n}v_t^{''(m)}$, $c'' = \hat{T}_e[\![e_2]\!]e\hat{n}v_t^{''(m)}$, and $a'' = \hat{T}_e[\![e_3]\!]e\hat{n}v_t^{''(m)}$.

   Then, $\hat{f}^{'(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots \lambda y_n.\langle p'_{(1)} \rhd c'_{(1)}, c'_{(2)}\rangle \bigsqcup \langle p'_{(1)} \rhd a'_{(1)}, a'_{(2)}\rangle\rangle \ldots\rangle\rangle$ and $\hat{f}^{''(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots \lambda y_n.\langle p''_{(1)} \rhd c''_{(1)}, c''_{(2)}\rangle \bigsqcup \langle p''_{(1)} \rhd a''_{(1)}, a''_{(2)}\rangle\rangle \ldots\rangle\rangle\rangle$. By the structural induction hypothesis and the definition of $\overset{t}{\sim}\sim$, it holds.

3. $e = e_1 e_2$: Let $f' = \hat{T}_e[\![e_1]\!]e\hat{n}v_t^{'(m)})$, $ap' = f_{(2)}(\hat{T}_e[\![e_2]\!]e\hat{n}v_t^{'(m)})$. Let $f'' = \hat{T}_e[\![e_1]\!]e\hat{n}v_t^{''(m)})$, $ap'' = f_{(2)}(\hat{T}_e[\![e_2]\!]e\hat{n}v_t^{''(m)})$.

   Then, $\hat{f}^{'(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots \lambda y_n.\langle f'_{(1)} \rhd ap'_{(1)}, ap'_{(2)}\rangle\rangle \ldots\rangle\rangle\rangle$, and $\hat{f}^{''(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots \lambda y_n.\langle f''_{(1)} \rhd ap''_{(1)}, ap''_{(2)}\rangle\rangle \ldots\rangle\rangle\rangle$. By the structural induction hypothesis and the definition of $\overset{t}{\sim}\sim$, it holds.

4. e = `lambda`$(x).e_1$: $\hat{f}^{'(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots \lambda y_n.\langle 1,\ \lambda y.\hat{T}_e[\![e_1]\!]\ e\hat{n}v_t^{'(m)}[x_i \mapsto y_i]\rangle\ \rangle \ldots\rangle\rangle\rangle$, and $\hat{f}^{''(m+1)} = \langle 1, \lambda y_1.\langle 1, \lambda y_2.\langle 1, \ldots \lambda y_n.\langle 1,\ \lambda y.\hat{T}_e[\![e_1]\!]\ e\hat{n}v_t^{''(m)}[x_i \mapsto y_i]\rangle\ \rangle \ldots\rangle\rangle\rangle$. $e\hat{n}v_t^{'(m)}[\![z]\!]\rangle \overset{t}{\sim}\sim e\hat{n}v_t^{''(m)}[\![z]\!]\rangle$, and by structural induction hypothesis, it holds.

□

The above lemma says that all possible monomorphic instances of a polymorphic function are related with respect to the order-of-demand property of all occurrences of their parameters and local objects. Based on this fact, we prove that both the global and local before analyses are polymorphically invariant.

Theorem 6.7 (Global Before Analysis) *Let f be a polymorphic function of arity n, and let f' and f'' be any two monomorphic instances of f. Assume that env' and env'' are abstract before escape semantic environments that map f' and f'' to elements of $\hat{D}_t$, respectively. Then, for X and Y,*

$$\mathsf{G_before?}(f', X, Y, env') = \mathsf{G_before?}(f'', X, Y, env'')$$

Proof : Let f'_a and f''_a be the auxiliary functions for f' and f'', respectively. From the definition of the global before test function,

$$\mathsf{G_before?}(f', X, Y, env') = (\hat{T}_e[\![f'_a\ x_{11}\ \ldots\ x_{no(n)}]\!]env'[f'_a \mapsto \hat{f}'_a, x_{ij} \mapsto y_{ij}])_{(1)}.$$

Then, by the definition of $\hat{T}_e$,

$$\mathsf{G_before?}(f', X, Y, env') = (\mathrm{NAP}_m(\hat{f}'_a, y'_{11}, \ldots, y'_{no(n)}))_{(1)}$$

where $m = \sum_i o(i)$, $\hat{f}' = \hat{T}_e[\![f']\!]env'$, $\hat{f}'_a = \hat{T}_e[\![f'_a]\!]env'[f' \mapsto \hat{f}']$, for each $x_{ij} \in X$, $y'_{ij} = \langle 2, W^{\tau'_i}\rangle$, for each $x_{ij} \in Y$, $y'_{ij} = \langle 0, W^{\tau'_i}\rangle$, and for each $x_{ij} \notin X \cup Y$, $y'_{ij} = \langle 1, W^{\tau'_i}\rangle$. Similarly,

$$\mathsf{G_before?}(f'', X, Y, env'') = (\mathrm{NAP}_m(\hat{f}''_a, y''_{11}, \ldots, y''_{no(n)}))_{(1)}$$

where $\hat{f}'' = \hat{T}_e[\![f'']\!]env''$, $\hat{f}''_a = \hat{T}_e[\![f''_a]\!]env''[f'' \mapsto \hat{f}'']$, for each $x_{ij} \in X$, $y''_{ij} = \langle 2, W^{\tau''_i}\rangle$, for each $x_{ij} \in Y$, $y''_{ij} = \langle 0, W^{\tau''_i}\rangle$, and for each $x_{ij} \notin X \cup Y$, $y''_{ij} = \langle 1, W^{\tau''_i}\rangle$. By Lemma 6.4, $\hat{f}' \overset{t}{\sim\sim} \hat{f}''$ and thus $\hat{f}'_a \overset{t}{\sim\sim} \hat{f}''_a$. By the definition of the worst-case escape function W, $y'_{ij} \overset{t}{\sim\sim} y''_{ij}$. Then, by the definition of $\overset{t}{\sim\sim}$, we have that

$$\mathrm{NAP}_m(\hat{f}'_a, y'_{11}, \ldots, y'_{no(n)})_{(1)} = \mathrm{NAP}_m(\hat{f}''_a, y''_{11}, \ldots, y''_{no(n)})_{(1)}.$$

Thus, we conclude that

$$\mathsf{G_before?}(f', X, Y, env') = \mathsf{G_before?}(f'', X, Y, env'')$$

□

Theorem 6.8 (Local Before Analysis) *Let f be a polymorphic function of arity n in an application $f\ e_1\ \ldots\ e_n$. Let f' and f'' be any two monomorphic instances of f, and e'_i and e''_i be two monomorphic instances of e_i. Assume that env' and env'' are abstract escape semantic environments that map f' and all free identifiers within e'_i, and f'' and all free identifiers within e''_i to elements of $\hat{D}_t$, respectively. Then, for X and Y,*

$$\mathsf{L_before?}(f', X, Y, e'_1, \ldots, e'_n, env') = \mathsf{L_before?}(f'', X, Y, e''_1, \ldots, e''_n, env'')$$

Proof : This can be proved in a similar way to the polymorphic invariance proof of the global before analysis. □

6.4 Analysis of Polymorphic Languages

The type variables appearing in the type of a `lambda`-bound identifier, called non-generic type variables, are shared in every occurrence of the identifier in the function body, and cannot be instantiated to different types in the same expression. However, the type variables which occur in the type of a `letrec`-bound identifier, called generic type variables, can be instantiated to different types in the same expression. Since the `letrec`-bound identifier is local to the expression, we know exactly how it is defined and can use this information to deal with each of its occurrences individually.

In proving the polymorphic invariance of the semantic analysis, we treated a polymorphic expression as the set of its possible monomorphic instances. In this section, we describe the semantic analysis of a higher-order functional language with a polymorphic type system which allows generic (parametric) polymorphism, and all type variables are

universally quantified at the top level. Furthermore, quantifiers can not be nested inside type expression, called shallow types. Occurrences of the same generic `letrec`-bound identifier may have different monotyped-instances. This can be resolved by making each of differently monotyped occurrences a distinct non-generic identifier. We transform each `letrec`-expression as follows:

`letrec`		`letrec`
$\quad x_1 = e_1;$		$\quad x_1^1 = e_1;\ \ldots\ x_1^{m(1)} = e_1;$
$\quad \vdots$	$\Longrightarrow$	$\quad \vdots$
$\quad x_n = e_n;$		$\quad x_1^n = e_n;\ \ldots\ x_n^{m(1)} = e_n;$
`in` $\ e$		`in` $\ e'$

where $m(i)$ is the number of occurrences of x_i with differently monomorphic instances in the expression e, and e' is derived from e by replacing the j^{th} occurrence of x_i with differently monomorphic instances by x_i^j for all $1 \leq i \leq n$ and $0 \leq j \leq m(i)$.

As an example, consider the polymorphic function as follows:

```
f x y z = letrec
             len a = if (null a) then 0 else 1 + len (cdr a);
             sum b = if (null b) then 0 else (car b) + sum (cdr b);
          in
             if (sum y)*(len y) < (sum (car z))*(len z)
             then x else (cdr x)
```

The types of functions `f`, `len`, and `sum` are

$$\texttt{f} \ : \ \forall\alpha.\ \alpha\ list \rightarrow int\ list \rightarrow int\ list\ list \rightarrow \alpha\ list$$
$$\texttt{len} \ : \ \forall\beta.\ \beta\ list \rightarrow int$$
$$\texttt{sum} \ : \ \forall\gamma.\ \gamma\ list \rightarrow int$$

Two occurrences of the `letrec`-bound identifier `len` are instantiated to different types, i.e.

The first occurrence of `len` : $\forall\beta.\ \beta\ list \rightarrow int\ [\ \beta = int\ ]$

The second occurrence of `len` : $\forall\beta.\ \beta\ list \rightarrow int\ [\ \beta = int\ list\ ]$

Thus, we transform the function definition of `f` into as follows:

```
f x y z = letrec
             len' a = if (null a) then 0 else 1 + len' (cdr a);
             len'' a = if (null a) then 0 else 1 + len'' (cdr a);
```

```
        sum b = if (null b) then 0 else (car b) + sum (cdr b);
    in
        if (sum y)*(len' y) < (sum (car z))*(len'' z)
        then x else (cdr x)
```

In order to get some polymorphically invariant information about the polymorphic function `f`, we find the simplest monomorphic instance of `f` and analyze that function using the monomorphic analysis. As a simplest monomorphic instance, we let $\alpha = int$. Then, the monotypes of functions `f`, `len'`, `len''` and `sum` are

`f` : $int\ list \rightarrow int\ list \rightarrow int\ list\ list \rightarrow int\ list$

`len'` : $int\ list \rightarrow int$

`len''` : $int\ list\ list \rightarrow int$

`sum` : $int\ list \rightarrow int$

Then, we apply the analysis for monomorphic languages to this transformed functions.

Chapter 7

Extensions to Non-strict Languages

So far in the previous chapters we have discussed a set of escape analyses for higher-order functional languages with *strict* semantics. Many modern functional languages, however, adopt a non-strict semantics and use the lazy evaluation model. These languages are more powerful than strict languages in expressiveness but requires quite a different kind of implementation model.

In this chapter, we extend the escape analysis and the reference escape analysis for a strict language to a non-strict language. First, we describe a method, based on a source-to-source transformation of non-strict programs, to extend the escape analysis and the reference escape analysis for a strict language to a non-strict language with *normal-order evaluation* using the analysis techniques for a strict language. The lazy evaluation model is identical to the normal-order evaluation model in the standard semantics, but not in an operational semantics. In lazy evaluation, arguments to a function are not evaluated unless and until their values are demanded, and are evaluated only once upon the first demand. Their values are then saved to be used for subsequent demands, thus avoiding reevaluation. Using the order-of-demand analysis described in Chapter 5, we extend the escape and reference escape analyses to non-strict functional languages using lazy evaluation. Finally, the extension for a non-strict language using optimized lazy evaluation based on the strictness information is discussed.

7.1 Escapement under Normal-Order Evaluation

Two methods are used for implementing non-strict semantics: either a normal-order (call-by-name) evaluation or a lazy (call-by-need) evaluation. In a non-strict semantics with normal order evaluation, all arguments to a function are passed to the function in an unevaluated form and are only evaluated whenever needed inside the function call. In order to achieve normal order evaluation, the following two mechanisms are required:

1. A mechanism to *delay* or *suspend* the evaluation of expressions when their evaluated values are not needed immediately;(for example, when expressions are passed to some non-strict function.)

2. A mechanism to *force* the evaluation of suspended expressions when their values are needed; (for example, when they are passed to some strict function.)

On a conventional machine, normal-order evaluation is effected by creating a closure, called *thunk*, consisting of some code along with bindings for its free variables, to represent a delayed expression. These thunks must generally be allocated in the heap and their evaluation requires a function call to force the evaluation.

In general, we can simulate a non-strict language using a strict language by the introduction of explicit delays and forces. A way to simulate normal-order evaluation is to insert some code to *delay* the evaluation of any expression which will not be immediately needed, including all arguments passed to user-defined functions, and to insert appropriate code to *force* the evaluation of the delayed expression when they are needed. It is possible to represent delayed expressions by using the same closures which are used for representing function values. Given an argument expression e, we can use beta-abstraction to translate e into the expression $(\lambda x.e)\ y$ for any y provided neither x nor y occur free in e. For convenience, we define as follows:

- `()` denotes a void identifier

- `[]` denotes a dummy expression.

Then we can write this as $(\lambda().e)$ [] without having to worry about naming problem. Whenever the value of e is required, we have to explicitly force the evaluation of e. To do this, we have to apply the closure for e to a dummy argument. This forcing is only required to be done when the value of the expression is actually needed. The delay and force operations can be implemented by a strict language as follows:

```
DELAY exp = lambda(). exp  /* create delayed expression */

FORCE delayed_exp = (delayed_exp [])  /* evaluation */
```

Thus, in general, any non-strict language can be simulated by a strict language through a source-to-source transformation of non-strict programs into their strict counterparts.

Since the operational semantics of a strict language is not equivalent to that of its non-strict counterpart, the escapement under non-strict semantics with normal-order evaluation becomes different from the escapement under strict semantics. As an example, consider the following non-strict (using normal-order evaluation) program:

```
letrec  f x y = x+y;
        g a = if (a=0) then (f a) else (f 3);
        h c d  = g (c+d);
in   ...
```

Consider the escapement of the parameters `c` and `d` from the function `h`. In a strict language, during the execution of `h`'s body, the expression `(c+d)` will be evaluated before `g` is applied to it, and therefore neither `c` or `d` escape from `h`. However, in a non-strict language, during the execution of `h`'s body, a closure representing the delayed expression `(c+d)` will be created and passed to `g`. During the execution of `g`'s body, the closure is passed to `f` and thus both `c` and `d` may escape from `h` within the closure representing the partial application of `f`.

Consider the escapement of references associated with occurrences of the parameters `c` and `d` from their function `h`. When `h` is called, its activation record contains two references, corresponding to the parameters `c` and `d`, to the actual arguments. The actual parameters are themselves closures. During the evaluation of the body of `h`, when `(g (c+d))` is evaluated, a closure is created for `(c+d)`. The references corresponding to `c` and `d` are copied into the closure. Then, the reference to this closure is copied into the activation record for `g`. During the execution of `g`'s body, this reference is passed to `f` and thus both references associated with `c` and `d` may escape from `h` within the closure representing the partial application of `f`.

Property 7.1 (Non-strict Escape Property) In the normal-order evaluation model, given an expression *exp*, the escape semantic value of *exp* when its evaluation is delayed, has the following properties:

- The first component of the escape semantic value of *exp* is the least upper bound of the first components of the escape semantic values of all identifiers in *exp*, because when *exp* is eventually evaluated, the values of all identifiers in *exp* will be needed.
- The second component of the escape semantic value of *exp* is err because the delayed *exp* itself is not a function type and thus it can not be applied directly to any argument.
- The escape semantic value of *exp* must be retained somehow in order to represent the escape semantic value when *exp* is evaluated.

7.1.1 Program Transformation

Thus, the escape analysis and the reference escape analysis for a non-strict functional language can be achieved by first transforming non-strict programs into equivalent strict programs and then by applying the analyses to the transformed programs. The source-to-source transformation function **N** of type $Exp \rightarrow \mathsf{Powerset}(Id) \rightarrow Exp$ for non-strict programs is defined in Figure 7.1.

7.1.2 Using Escape Analysis for Strict Languages

The abstract escape semantic function $\hat{O}_c$ for a dummy expression [] is defined as follows:

$$\hat{O}_c[\![[]]\!] \quad = \quad \langle 0, err \rangle$$

Global Non-strict Escape Test

We describe a global escape test for non-strict functions. Given a non-strict function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position i of an interesting parameter, and an abstract escape semantic environment $e\hat{n}v_o$ mapping a transformed version of f to an element of $\hat{D}_o$, the global non-strict escape test function G_n_escape? determines whether the i^{th} parameter of the non-strict function f could possibly escape f globally or not. It is defined as follows:

$$\mathsf{G_n_escape?}(f, i, e\hat{n}v_o) = (\hat{O}_e[\![f'\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_o[x_i \mapsto y_i])_{(1)}$$

where f' is a transformed version of f,

$$y_i = \langle 1, \lambda().\langle 1, W^{\tau_i} \rangle \rangle,$$

for all $j \leq n$ and $j \neq i$,

$$y_j = \langle 0, \lambda().\langle 0, W^{\tau_i} \rangle \rangle,$$

$$
\begin{array}{lcl}
\mathbf{N}[\![c]\!]\ FId & \Rightarrow & \lambda().c \\
\mathbf{N}[\![x]\!]\ FId & \Rightarrow & \text{if } (x \in FId) \text{ then } x \text{ else } (x\ \texttt{[]}) \\
\mathbf{N}[\![e_1 + e_2]\!]\ FId & \Rightarrow & (\mathbf{N}[\![e_1]\!]\ FId) + (\mathbf{N}[\![e_2]\!]\ FId) \\
 & & \text{/* same for strict operators like } (e_1 - e_2) \text{ and } (e_1 = e_2) \text{ */} \\
\mathbf{N}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\ FId & \Rightarrow & \texttt{if}\ (\mathbf{N}[\![e_1]\!]\ FId) \\
 & & \texttt{then}\ (\mathbf{N}[\![e_2]\!]\ FId) \\
 & & \texttt{else}\ (\mathbf{N}[\![e_3]\!]\ FId) \\
\mathbf{N}[\![e_1 e_2]\!]\ FId & \Rightarrow & \text{if } e_1 \in \{\ \texttt{car, cdr}\ \} \\
 & & \text{then /* strict operator */} \\
 & & (\mathbf{N}[\![e_1]\!]\ FId)\ \ (\mathbf{N}[\![e_2]\!]\ FId) \\
 & & \text{else /* non-strict operator */} \\
 & & (\mathbf{N}[\![e_1]\!]\ FId)\ \ (\lambda().\mathbf{N}[\![e_2]\!]\ FId) \\
\mathbf{N}[\![\texttt{lambda}(x).e]\!]\ FId & \Rightarrow & \texttt{lambda}(x).(\mathbf{N}[\![e]\!]\ FId) \\
\mathbf{N}[\![\texttt{letrec}\ x_1 = e_1; \ldots x_n = e_n; \texttt{in}\ e]\!]\ FId & \Rightarrow & \texttt{letrec} \\
 & & x_1 = \mathbf{N}[\![e_1]\!]\ (FId \cup \{x_1, \ldots, x_n\}); \\
 & & \ldots \\
 & & x_n = \mathbf{N}[\![e_n]\!]\ (FId \cup \{x_1, \ldots, x_n\}); \\
 & & \texttt{in} \\
 & & \mathbf{N}[\![e_1]\!]\ (FId \cup \{x_1, \ldots, x_n\}) \\
 & & \\
\mathbf{N}_P[\![pr]\!] & \Rightarrow & \mathbf{N}[\![pr]\!]\ \emptyset
\end{array}
$$

Figure 7.1: Non-strict Program Transformation

and τ_i is the type of the i^{th} parameter of f. Then, from the result of the global non-strict escape test function, we can conclude as follows:

- If G_n_escape?$(f, i, e\hat{n}v_o) = 0$ then we conclude that in any possible application of a non-strict f to n arguments, the i^{th} delayed argument *does not* escape f.
- If G_n_escape?$(f, i, e\hat{n}v_o) = 1$ then it means that the i^{th} delayed argument *could* escape f in some possible application of a non-strict f to n.

Local Non-strict Escape Test

We describe a local escape test for non-strict functions in a particular context. Given a non-strict function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in an application $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and an abstract escape semantic environment $e\hat{n}v_o$ mapping a transformed version of f and the free identifiers within e_1 through e_n to elements of $\hat{D}_o$, the local non-strict escape test function L_n_escape? determines whether the i^{th} parameter of the non-strict function f could escape f during the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$\text{L_n_escape?}(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = (\hat{O}_e[\![f'\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_o[x_i \mapsto y_i])_{(1)}$$

where f' is the transformed version of f,

$$y_i = \langle 1, \lambda().\langle 1, (\hat{O}_e[\![e_i]\!]\ e\hat{n}v_o)_{(2)}\rangle\rangle,$$

and for all $j \leq n$ and $j \neq i$,

$$y_j = \langle 0, \lambda().\langle 0, (\hat{O}_e[\![e_j]\!]\ e\hat{n}v_o)_{(2)}\rangle\rangle.$$

Then, from the result of the local non-strict escape test function, we can conclude the following:

- If L_n_escape?$(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = 0$ then we conclude that the i^{th} delayed argument *does not* escape f locally in the particular application of f to e_1 through e_n.
- If L_n_escape?$(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = 1$ then it means that the i^{th} delayed argument *could* escape f locally in the particular application of f to e_1 through e_n.

7.1.3 Using Reference Escape Analysis for Strict Languages

The abstract reference escape semantic function $\hat{R}_c$ for a dummy expression [] is defined as follows:

$$\hat{K}_r[\![[]]\!] \quad = \quad \langle\langle 0, 0\rangle, err\rangle$$

Global Non-strict Reference Escape Test

Given a non-strict function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position (i,j) of an interesting occurrence of a parameter, and an abstract reference escape semantic environment $e\hat{n}v_r$ mapping the auxiliary function of the transformed version of f to an element of $\hat{D}_r$, the global non-strict reference escape test function G_n_refescape? determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of the non-strict function f could escape f globally. It is defined as follows:

$$\mathsf{G_n_refescape?}(f,i,j,e\hat{n}v_r) =$$
$$(\hat{R}_e[\![f''\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\hat{n}v_r[f'' \mapsto \hat{f''}, x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f'' is an auxiliary function for f', f' is a transformed function of f,

$$\hat{f''} = \hat{R}_e[\![f']\!]e\hat{n}v_r,$$

$$y_{ij} = \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, W^{\tau_i}\rangle\rangle,$$

for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{\tau_i}\rangle\rangle,$$

for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{\tau_i}\rangle\rangle,$$

and τ_i is the type of the i^{th} parameter of f. Then, from the result of the global non-strict reference escape test function, we can conclude as follows:

- If $\mathsf{G_n_refescape?}(f,i,j,e\hat{n}v_r) = 0$ then we conclude that in any possible application of a non-strict function f to n arguments, the reference associated with the j^{th} occurrence of the i^{th} delayed argument *does not* escape f.

- If $\mathsf{G_n_refescape?}(f,i,j,e\hat{n}v_r) = 1$ then it means that in some possible application of a non-strict function f to n arguments, the reference associated with the j^{th} occurrence of the i^{th} delayed argument *could* escape f.

Local Non-strict Reference Escape Test

Given a non-strict function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in a particular function application $f\ e_1\ \ldots\ e_n$, the position (i,j) of an interesting occurrence of a parameter, and an abstract reference escape semantic environment $e\hat{n}v_r$ mapping the auxiliary function for

the transformed version of f and the free identifiers within e_1 through e_n to elements of $\hat{D}_r$, the global non-strict reference escape test function L_n_refescape? determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of the non-strict function f could escape f in the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$\begin{aligned}&\mathsf{L_n_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = \\ &\quad (\hat{R}_e[\![f''\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\hat{n}v_r[f'' \mapsto \hat{f}'', x_{ij} \mapsto y_{ij}])_{(1)(1)}\end{aligned}$$

where f'' is an auxiliary function for f', f' is a transformed function of f,

$$\hat{f}'' = \hat{R}_e[\![f']\!]e\hat{n}v_r,$$

$$y_{ij} = \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, (\hat{R}_e[\![e_i]\!]e\hat{n}v_r)_{(2)}\rangle\rangle,$$

for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, (\hat{R}_e[\![e_i]\!]e\hat{n}v_r)_{(2)}\rangle\rangle,$$

and for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, (\hat{R}_e[\![e_l]\!]e\hat{n}v_r)_{(2)}\rangle\rangle.$$

Then, from the result of the local non-strict reference escape test function, we can conclude the following:

- If $\mathsf{L_n_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = 0$ then we conclude that the reference associated with the j^{th} occurrence of the i^{th} delayed argument *does not* escape f in $(f\ e_1\ \ldots\ e_n)$.
- If $\mathsf{L_n_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = 1$ then it means that the reference associated with the j^{th} occurrence of the i^{th} delayed argument *could* escape f in $(f\ e_1\ \ldots\ e_n)$.

7.1.4 Examples

As an example, consider the following non-strict program:

```
letrec  f x y = x+y;
        g a = if (a=0) then (f a) else (f 3);
        h c d  = g (c+d);
in   ...
```

where `f, g, h` : $int \rightarrow int \rightarrow int$. The transformed functions `f'`, `g'` and `h'` for `f`, `g` and `h`, based on the program transform function **N**, are defined as follows:

```
letrec  f' x y = (x []) + (y []);
        g' a = if ((a []) = 0) then (f' lambda().(a []))
               else (f' lambda().3);
        h' c d = g' lambda().((c []) + (d []));
in   ...
```

where `f', g', h'` : $(* \to int) \to (* \to int) \to (* \to int)$ and * denotes the type of a void identifier and a dummy expression.

Non-strict Escape Information

The definitions of the escape semantic values f', g', and h' of `f'`, `g'`, and `h'` are expressed as follows:

$$\begin{aligned}
f' &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\langle 0, err\rangle\rangle\rangle \\
g' &= \langle 0, \lambda a.(f'_{(2)}\ \langle a_{(1)}, \lambda().a\ \langle 0, err\rangle) \sqcup (f'_{(2)}\ \langle 0, \lambda().\langle 0, err\rangle\rangle)\rangle \\
&= \langle 0, \lambda a.(f'_{(2)}\ \langle a_{(1)}, \lambda().a\ \langle 0, err\rangle)\rangle \\
h' &= \langle 0, \lambda c.\langle c_{(1)}, \lambda d.g'_{(2)}\ \langle (c_{(1)} \sqcup d_{(1)}), \lambda().\langle 0, err\rangle\rangle\rangle\rangle
\end{aligned}$$

Let $e\hat{n}v_o = [\texttt{f'} \mapsto f', \texttt{g'} \mapsto g', \texttt{h'} \mapsto h']$.

To find the global escape property of the non-strict function `h`, we apply the non-strict global escape test function G_n_escape? as follows:

$$\begin{aligned}
\texttt{G_n_escape?}(\texttt{h}, 1, e\hat{n}v_o) &= (\hat{O}_e[\![\texttt{h' c d}]\!]e\hat{n}v'_o)_{(1)} \\
&= 1
\end{aligned}$$

where

$$e\hat{n}v'_o = e\hat{n}v_o[\texttt{c} \mapsto \langle 1, \lambda().\langle 1, W^{(*\to int)\to(*\to int)}\rangle\rangle, \texttt{d} \mapsto \langle 0, \lambda().\langle 0, W^{(*\to int)\to(*\to int)}\rangle\rangle]$$

and

$$W^{(*\to int)\to(*\to int)} = \lambda x.\langle x_{(1)}, \lambda y.x_{(1)} \sqcup y_{(1)}, err\rangle\rangle.$$

Then, we can conclude that the first parameter `c` of the function `h` could escape `h` globally. Similarly,

$$\begin{aligned}
\texttt{G_n_escape?}(\texttt{h}, 2, e\hat{n}v_o) &= (\hat{O}_e[\![\texttt{h' c d}]\!]e\hat{n}v'_o)_{(1)} \\
&= 1
\end{aligned}$$

where

$$e\hat{n}v'_o = e\hat{n}v_o[\texttt{c} \mapsto \langle 0, \lambda().\langle 0, W^{(*\to int)\to(*\to int)}\rangle\rangle, \texttt{d} \mapsto \langle 1, \lambda().\langle 1, W^{(*\to int)\to(*\to int)}\rangle\rangle]$$

and

$$W^{(*\to int)\to(*\to int)} = \lambda x.\langle x_{(1)}, \lambda y.x_{(1)} \sqcup y_{(1)}, err\rangle\rangle.$$

Thus, we can conclude that the second parameter `d` of the function `h` could also escape `h`.

Non-strict Reference Escape Information

The definitions of the reference escape semantic values f', g', and h' of `f'`, `g'`, and `h'` are expressed as follows:

$$\begin{aligned}
f' &= \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
g' &= \langle\langle 0,0\rangle, \lambda a.(f'_{(2)}\ \langle a_{(1)}, \lambda().a\ \langle\langle 0,0\rangle, err\rangle\rangle \sqcup (f'_{(2)}\ \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, err\rangle\rangle\rangle\rangle \\
&= \langle\langle 0,0\rangle, \lambda a.(f'_{(2)}\ \langle a_{(1)}, \lambda().a\ \langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
h' &= \langle\langle 0,0\rangle, \lambda c.\langle c_{(1)}, \lambda d.g'_{(2)}\ \langle (c_{(1)} \sqcup d_{(1)}), \lambda().\langle\langle 0,0\rangle, err\rangle\rangle\rangle\rangle
\end{aligned}$$

The auxiliary functions `f''`, `g''` and `h''` for the transformed functions `f'`, `g'` and `h'` are defined as follows:

```
f'' x y = (x []) + (y []);
g' a1 a2 = if ((a1 []) = 0) then (f' lambda().(a2 []))
               else (f' lambda().3);
h' c d = g' lambda().((c []) + (d []));
```

Then, the definitions of reference escape semantic values f'', g'', and h'' of `f''`, `g''`, and `h''` are expressed as follows:

$$\begin{aligned}
f'' &= \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
g'' &= \langle\langle 0,0\rangle, \lambda a1.\langle a1_{(1)}, \lambda a2.(f'_{(2)}\ \langle a2_{(1)}, \lambda().a2_{(2)}\ \langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
h'' &= \langle\langle 0,0\rangle, \lambda c.\langle c_{(1)}, \lambda d.g'_{(2)}\ \langle (c_{(1)} \sqcup d_{(1)}), \lambda().\langle\langle 0,0\rangle, err\rangle\rangle\rangle\rangle
\end{aligned}$$

Let $e\hat{n}v_r = [\texttt{f'} \mapsto f', \texttt{f''} \mapsto f'', \texttt{g'} \mapsto g', \texttt{g''} \mapsto g'', \texttt{h'} \mapsto h', \texttt{h''} \mapsto h'']$.

To find the global reference escape property of the non-strict function `h`, we apply the non-strict global reference escape test function G_n_refescape? as follows:

$$\begin{aligned}
\text{G_n_refescape?}(\texttt{h}, 1, 1, e\hat{n}v_r) &= (\hat{R}_e[\![\texttt{h'' c d}]\!]e\hat{n}v'_r)_{(1)(1)} \\
&= 1
\end{aligned}$$

where

$$\begin{aligned}
e\hat{n}v'_r = e\hat{n}v_r[&\texttt{c} \mapsto \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, W^{(*\to int)\to(*\to int)}\rangle\rangle, \\
&\texttt{d} \mapsto \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{(*\to int)\to(*\to int)}\rangle\rangle]
\end{aligned}$$

Then, we can conclude that the reference associated with the first occurrence of the first parameter `c` of the function `h` could escape `h` globally. Similarly,

$$\begin{aligned} \mathsf{G_n_refescape?}(\mathtt{h}, 2, 1, \hat{en}v_r) &= (\hat{R}_e[\![\mathtt{h''\ c\ d}]\!]\hat{en}v'_r)_{(1)(1)} \\ &= 1 \end{aligned}$$

where

$$\begin{aligned} \hat{en}v'_r = \hat{en}v_r[\mathtt{c} \mapsto \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{(*\to int)\to(*\to int)}\rangle\rangle, \\ \mathtt{d} \mapsto \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, W^{(*\to int)\to(*\to int)}\rangle\rangle] \end{aligned}$$

Thus, we can conclude that the reference associated with the first occurrence of the second parameter `d` of the function `h` could also escape `h`.

7.2 Escapement under Lazy Evaluation

The demand-driven evaluation of an expression in a non-strict language is effected by creating some delayed representation which contains enough information to enable the expression to be evaluated later when needed. In particular, this requires some mechanism for creating the delayed expression and some mechanism for forcing its evaluation. A naive implementation of normal-order reduction would result in an implementation in which actual parameters were copied as they were bounded to formal parameters in function application. Using this strategy, an argument could be evaluated several different times, resulting in some inefficiency. The lazy evaluation model is an efficient implementation method in which each argument is computed at most once. In a non-strict semantics using lazy evaluation, all arguments to a function are passed to the function in an unevaluated form and are evaluated only once when first needed inside the function call. Thus, in lazy evaluation, some other mechanism is needed to avoid the recomputation of a value. In addition, the representation of the delayed expression must contain the code to be evaluated and the environment in which to evaluate the code. In order to effect the lazy evaluation, we basically require the following three mechanisms:

1. A mechanism to *delay* or *suspend* the evaluation of expressions when their evaluated values are not needed immediately;(for example, when expressions are passed to some non-strict function.)

2. A mechanism to *force* the evaluation of evaluation-suspended expressions when the evaluated values of evaluation-suspended expressions are needed; (for example, when they are passed to some strict function.)

3. A mechanism to *avoid* the re-evaluation of suspended expressions when their values are needed and their evaluations have already been forced.

On a conventional machine, lazy evaluation is effected by creating a run-time structure, called *self-modifying thunk*, consisting of some code along with bindings for its free variables, the status of the evaluation and the value when already evaluated, to represent a delayed expression. These self-modifying thunks must generally be allocated in the heap and their evaluation requires a test of the evaluation status, a function call to force the evaluation, and an update with the evaluated value. Due to the absence of side effects of functional languages, lazy evaluation is equivalent to normal-order reduction under the standard semantics, but it is often more efficient since it requires no recomputation. We can not implement a lazy evaluation model using a strict pure functional language because we cannot explicitly overwrite the environment. Lazy evaluation can be implemented only through the use of side-effects. One way of implementing the lazy evaluation model is as follows:

```
DELAY exp = lambda(). exp  /* create delayed expression */

FORCE delayed_exp = if already_evaluated?(delayed_exp)
                    then return V;  /* no evaluation */
                    else V := (delayed_exp []);  /* evaluation */
                         save V in some place;
                         return V;
```

The escapement under a non-strict semantics with lazy evaluation is not equivalent to the escapement under a non-strict semantics with normal-order evaluation, because their operational semantics are different. Consider the following program as before but with lazy evaluation:

```
letrec  f x y = x+y;
        g a = if (a=0) then (f a) else (f 3);
        h c d  = g (c+d);
in   ...
```

Consider the escapement of the parameters `c` and `d` from their function `h`. As described before, in a non-strict language with normal-order evaluation, a closure representing the

delayed expression of `(c+d)` will be created and passed to `g` during the execution of `h`'s body. During the execution of `g`'s body, this closure is passed to `f` and thus both `c` and `d` may escape from `h` within the closure representing the partial application of `f`. However, in a non-strict language with lazy evaluation, a closure representing the delayed expression of `(c+d)` will be created and passed to `g` during execution of `h`'s body. In the execution of `g`, the delayed expression of `(c+d)` is forced to be evaluated during the evaluation of `(a=0)` and thus neither `c` and nor `d` will escape `h`.

Consider the escapement of references associated with occurrences of the parameters `c` and `d` from their function `h`. When `h` is called, its activation record contains two references, corresponding to the parameters `c` and `d`, to actual parameters which are self-modifying thunks. During the evaluation of the body of `h`, when `(g (c+d))` is evaluated, a self-modifying thunk is created for the delayed evaluation of `(c+d)`. The references corresponding to `c` and `d` are copied into the self-modifying thunk. Then, the reference to this self-modifying thunk is copied into the activation record for `g`. During the execution of `g`'s body, this self-modifying thunk is actually evaluated via `(a=0)` and the result value is saved in the self-modifying thunk for later demand. When the value of this self-modifying thunk is demanded in future, the value saved is just returned without further evaluation. Thus, even if this self-modifying thunk is passed to `f` in the partial application to `f`, both references associated with `c` and `d`, which are contained in the self-modifying thunk, do not escape from `h` because they are no longer needed.

Property 7.2 (Lazy Escape Property) In the lazy evaluation model, given an expression *exp*, the escape semantic value of *exp* when its evaluation is delayed, has the following properties:

- The first component of the escape semantic value of *exp* is the least upper bound of
  1. the least upper bound of the first components of the escape semantic values of all identifiers in *exp* that have not yet been evaluated.
  2. the least upper bound of the first components of the escape semantic values of all identifiers in *exp* that have been already evaluated, because when *exp* is eventually evaluated the values of all identifiers in *exp* are needed.
- The second component of the escape semantic value of *exp* is err because *exp* itself is not a function type and thus it can not be applied directly to any argument.
- The escape semantic value of *exp* must be retained somehow in order to represent the escape semantic value when *exp* is evaluated.

Thus, information about evaluation status is useful for the escape analysis and the reference escape analysis of a non-strict functional language with lazy evaluation.

7.2.1 Using Status-of-Evaluation Information

Information on the evaluation status of arguments to a function when they are demanded is useful for describing the escapement behavior under lazy evaluation. Given an occurrence x_i of a variable x in the body of a function f, if there exists another occurrence x_j of x such that x_j is demanded before x_i in each possible execution of the body of f then we say that x_i has an *evaluated status*. This means that x must already have been evaluated by the time x_i is encountered. Information about the evaluated status of an occurrence of a parameter of a function can be inferred at compile-time using the order-of-demand analysis described in Chapter 5.

For example, consider the following function:

```
g a = if (a=0) then (f a) else (f 3);
```

Let the two occurrences of the parameter `a` in the body of the function `g` be `a'` and `a''` from left-to-right, respectively. The occurrence `a''` has an evaluated status, because the argument `a` is evaluated before it is demanded via `a''` due to the demand via the occurrence `a'`.

7.2.2 Escape Analysis for Lazy Evaluation

The abstract escape semantic domain $\hat{D}_o$ (an abstraction of D_o), and the domain $\hat{E}_o$ of abstract escape environments are defined as follows:

$$\hat{D}_o = \sum_{\tau} \hat{D}_o^{\tau} \qquad \text{/* Abstract escape semantic domain */}$$

$$\hat{E}_o = Id \rightarrow \hat{D}_o \qquad \text{/* Domain of abstract escape environments */}$$

$$\begin{array}{lcll} \hat{D}_o^{int} & = & \hat{B}_o \times \{err\} & \text{abstract subdomain for integers} \\ \hat{D}_o^{bool} & = & \hat{B}_o \times \{err\} & \text{abstract subdomain for booleans} \\ \hat{D}_o^{\tau_1 \rightarrow \tau_2} & = & \hat{B}_o \times (\hat{D}_o^{\tau_1} \rightarrow \hat{D}_o^{\tau_2}) & \text{abstract subdomain for functions} \\ \hat{D}_o^{\tau_1\ list} & = & \hat{D}_o^{\tau_1} & \text{abstract subdomain for lists} \end{array}$$

The abstract escape semantic functions are as follows:

$$\begin{array}{lcll} \bar{O}_c & : & Con \rightarrow \hat{D}_o & \text{/* Abstract escape semantic function for constants */} \\ \bar{O}_e & : & Exp \rightarrow \hat{E}_o \rightarrow \hat{D}_o & \text{/* Abstract escape semantic function for expressions */} \\ \bar{O}_{pr} & : & Program \rightarrow \hat{D}_o & \text{/* Abstract escape semantic function for programs */} \end{array}$$

$$
\begin{array}{lcl}
\bar{O}_c[\![[]]\!] & = & \langle 0, err\rangle \\
\bar{O}_c[\![c]\!] & = & \langle 0, err\rangle, \;\; c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\} \\
\bar{O}_c[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (The bottom element of } \hat{D}_o.) \\
\bar{O}_c[\![\texttt{cons}]\!] & = & \langle 0, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\
\bar{O}_c[\![\texttt{car}]\!] & = & \langle 0, \lambda x.x\rangle \\
\bar{O}_c[\![\texttt{cdr}]\!] & = & \langle 0, \lambda x.x\rangle \\
\bar{O}_c[\![\texttt{null}]\!] & = & \langle 0, \lambda x.\langle 0, err\rangle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\bar{O}_e[\![c]\!]e\hat{n}v_o & = & \bar{O}_c[\![c]\!] \\
\bar{O}_e[\![x]\!]e\hat{n}v_o & = & e\hat{n}v_o[\![x]\!] \\
\bar{O}_e[\![e_1 + e_2]\!]e\hat{n}v_o & = & \langle 0, err\rangle \text{ /* same for } e_1 - e_2 \text{ and } e_1 = e_2 \text{ */} \\
\bar{O}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\hat{n}v_o & = & (\bar{O}_e[\![e_1]\!]e\hat{n}v_o) \sqcup (\bar{O}_e[\![e_2]\!]e\hat{n}v_o) \\
\bar{O}_e[\![e_1e_2]\!]e\hat{n}v_o & = & (\bar{O}_e[\![e_1]\!]e\hat{n}v_o)_{(2)}\ (\bar{O}_e[\![e_2]\!]e\hat{n}v_o) \\
\bar{O}_e[\![\texttt{lambda}(x).e]\!]e\hat{n}v_o & = & \langle \bar{V}, \lambda y.\bar{O}_e[\![e]\!]e\hat{n}v_o[x \mapsto y]\rangle
\end{array}
$$

where

$\bar{V} = 0 \sqcup (\bigsqcup_{z \in F - F^{eval}} (e\hat{n}v_o[\![z]\!])_{(1)}) \sqcup (\bigsqcup_{z \in F^{eval}} (\bar{O}_e[\![\ (z\ [])\ ]\!]e\hat{n}v_o)_{(1)})$,

F = Set of free identifiers in $(\texttt{lambda}(x).e)$, and

F^{eval} = Set of free identifiers with an *evaluated status*.

$$
\bar{O}_e[\![\texttt{letrec } x_1 = e_1; \ldots x_n = e_n; \texttt{ in } e]\!]e\hat{n}v_o \;\; = \;\; \bar{O}_e[\![e]\!]e\hat{n}v'_o
$$

where $e\hat{n}v'_o = e\hat{n}v_o[x_1 \mapsto \bar{O}_e[\![e_1]\!]e\hat{n}v'_o, \ldots, x_n \mapsto \bar{O}_e[\![e_n]\!]e\hat{n}v'_o]$

$$
\bar{O}_{pr}[\![pr]\!] \;\; = \;\; \bar{O}_e[\![pr]\!]nul\hat{l}env_o
$$

Figure 7.2: Abstract Escape Semantic Functions for Lazy Evaluation

The definitions of the abstract escape semantic functions are given in Figure 7.2. Note that the definition of the abstract escape semantic function $\bar{O}_e$ for lambda expressions of the form $\texttt{lambda}(x).e$ treats free identifiers differently according to their evaluation status.

Global Lazy Escape Test

We describe a global escape test for lazy functions. Given a lazy function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position i of an interesting parameter, an abstract escape semantic environment $e\hat{n}v_o$ mapping a transformed version of f to an element of $\hat{D}_o$, the global lazy escape test function $\mathsf{G_l_escape?}$ determines whether the i^{th} parameter of the lazy function f could possibly escape f globally or not. It is defined as follows:

$$\mathsf{G_l_escape?}(f, i, e\hat{n}v_o) = (\bar{O}_e[\![f'\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_o[x_i \mapsto y_i])_{(1)}$$

where f' is a transformed function of f,

$$y_i = \langle 1, \lambda().\langle 1, W^{\tau_i}\rangle\rangle,$$

for all $j \leq n$ and $j \neq i$,

$$y_j = \langle 0, \lambda().\langle 0, W^{\tau_i}\rangle\rangle,$$

and τ_i is the type of the i^{th} parameter of f. Then, from the result of the global lazy escape test function, we can conclude as follows:

- If $\mathsf{G_l_escape?}(f, i, e\hat{n}v_o) = 0$ then we conclude that in any possible application of a lazy function f to n arguments, the i^{th} delayed argument *does not* escape f.
- If $\mathsf{G_l_escape?}(f, i, e\hat{n}v_o) = 1$ then it means that the i^{th} delayed argument *could* escape f in some possible application of a lazy function f to n.

Local Lazy Escape Test

We describe a local escape test for lazy functions in a particular context. Given a lazy function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in an application context $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and an abstract escape semantic environment $e\hat{n}v_o$ mapping a transformed version of f and the free identifiers within e_1 through e_n to elements of $\hat{D}_o$, the local lazy escape test function $\mathsf{L_l_escape?}$ determines whether the i^{th} parameter of the lazy function f could escape f in the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$\mathsf{L_J_escape?}(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = (\bar{O}_e[\![f'\ x_1\ \ldots\ x_n]\!]\ e\hat{n}v_o[x_i \mapsto y_i])_{(1)}$$

where f' is a transformed function of f,

$$y_i = \langle 1, \lambda().\langle 1, (\bar{O}_e[\![e_i]\!]\ e\hat{n}v_o)_{(2)}\rangle\rangle$$

and for all $j \leq n$ and $j \neq i$,

$$y_j = \langle 0, \lambda().\langle 0, (\bar{O}_e[\![e_j]\!]\ e\hat{n}v_o)_{(2)}\rangle\rangle.$$

Then, from the result of the local lazy escape test function, we can conclude the following:

- If $\mathsf{L_J_escape?}(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = 0$ then we conclude that the i^{th} delayed argument *does not* escape f in the particular application of f to e_1 through e_n.
- If $\mathsf{L_J_escape?}(f, i, e_1, \ldots, e_n, e\hat{n}v_o) = 1$ then it means that the i^{th} delayed argument *could* escape f in the particular application of f to e_1 through e_n.

7.2.3 Reference Escape Analysis for Lazy Evaluation

The abstract reference escape semantic domain $\hat{D}_r$ and the domain $\hat{E}_r$ of the abstract reference escape environments are defined as follows:

$$\begin{array}{lcll}
\hat{D}_r & = & \sum_{\tau} \hat{D}_r^{\tau} & \text{/* Abstract reference escape semantic domain */} \\
\hat{E}_r & = & Id \rightarrow \hat{D}_r & \text{/* Domain of abstract reference escape environments */}
\end{array}$$

$$\begin{array}{lcll}
\hat{D}_r^{int} & = & \hat{B}_r \times \{err\} & \text{abstract subdomain for integers} \\
\hat{D}_r^{bool} & = & \hat{B}_r \times \{err\} & \text{abstract subdomain for booleans} \\
\hat{D}_r^{\tau_1 \rightarrow \tau_2} & = & \hat{B}_r \times (\hat{D}_r^{\tau_1} \rightarrow \hat{D}_r^{\tau_2}) & \text{abstract subdomain for functions} \\
\hat{D}_r^{\tau\ list} & = & \hat{D}_r^{\tau} & \text{abstract subdomain for lists of type} \tau\ list
\end{array}$$

We now introduce the abstract reference escape semantic functions to give the syntax the reference escape meaning as follows:

$$\begin{array}{lcll}
\bar{R}_c & : & Con \rightarrow \hat{D}_r & \text{/* Abstract reference escape function for constants */} \\
\bar{R}_e & : & Exp \rightarrow \hat{E}_r \rightarrow \hat{D}_r & \text{/* Abstract reference escape function for expressions */} \\
\bar{R}_{pr} & : & Program \rightarrow \hat{D}_r & \text{/* Abstract reference escape function for programs */}
\end{array}$$

The semantic equations for the abstract reference escape semantic functions are given in Figure 7.3. Note that the definition of the abstract reference escape semantic function $\bar{R}_e$ for lambda expressions of the form $\texttt{lambda}(x).e$ treats free identifiers differently according to their evaluation status.

$$
\begin{array}{lcl}
\bar{R}_c[\![[]]\!] & = & \langle\langle 0,0\rangle, err\rangle \\
\bar{R}_c[\![c]\!] & = & \langle\langle 0,0\rangle, err\rangle, \;\; c \in \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}\} \\
\bar{R}_c[\![\texttt{nil}^{\tau\, list}]\!] & = & \bot_\tau \text{ (The bottom element of } \hat{D}_r.) \\
\bar{R}_c[\![\texttt{cons}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\
\bar{R}_c[\![\texttt{car}^{\texttt{s}}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\mathrm{cut}^{\mathrm{s}}(x)\rangle \\
\bar{R}_c[\![\texttt{cdr}]\!] & = & \langle\langle 0,0\rangle, \lambda x.x\rangle \\
\bar{R}_c[\![\texttt{null}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle\langle 0,0\rangle, err\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\bar{R}_e[\![c]\!]e\hat{n}v_r & = & \bar{R}_c[\![c]\!] \\
\bar{R}_e[\![x]\!]e\hat{n}v_r & = & e\hat{n}v_r[\![x]\!] \\
\bar{R}_e[\![e_1 + e_2]\!]e\hat{n}v_r & = & \langle\langle 0,0\rangle, err\rangle \text{ /* same for } e_1 - e_2 \text{ and } e_1 = e_2 \text{ */} \\
\bar{R}_e[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]e\hat{n}v_r & = & (\bar{R}_e[\![e_1]\!]e\hat{n}v_r) \sqcup (\bar{R}_e[\![e_2]\!]e\hat{n}v_r) \\
\bar{R}_e[\![e_1 e_2]\!]e\hat{n}v_r & = & (\bar{R}_e[\![e_1]\!]e\hat{n}v_r)_{(2)}\, (\bar{R}_e[\![e_2]\!]e\hat{n}v_r) \\
\bar{R}_e[\![\texttt{lambda}(x).e]\!]e\hat{n}v_r & = & \langle \bar{V}, \lambda y.\bar{R}_e[\![e]\!]e\hat{n}v_r[x \mapsto y]\rangle
\end{array}
$$

where

$$\bar{V} = \langle 0,0\rangle \sqcup \Big(\bigsqcup_{z \in F - F^{eval}} (e\hat{n}v_r[\![z]\!])_{(1)}\Big) \sqcup \Big(\bigsqcup_{z \in F^{eval}} (\bar{O}_e[\![\,(z\;[])\,]\!]e\hat{n}v_r)_{(1)}\Big)$$

F = Set of free identifiers in $(\texttt{lambda}(x).e)$, and

F^{eval} = Set of free identifiers with an *evaluated status*.

$$\bar{R}_e[\![\texttt{letrec } x_1 = e_1; \ldots x_n = e_n; \texttt{in } e]\!]e\hat{n}v_r \;\; = \;\; \bar{R}_e[\![e]\!]e\hat{n}v'_r$$

where $e\hat{n}v'_r = e\hat{n}v_r[x_1 \mapsto \bar{R}_e[\![e_1]\!]e\hat{n}v'_r, \ldots, x_n \mapsto \bar{R}_e[\![e_n]\!]e\hat{n}v'_r]$

$$\bar{R}_{pr}[\![pr]\!] \;\; = \;\; \bar{R}_e[\![pr]\!]nul\hat{l}env_r$$

Figure 7.3: Abstract Reference Escape Semantic Functions for Lazy Evaluation

Global Lazy Reference Escape Test

Given a lazy function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n, the position (i,j) of an interesting occurrence of a parameter, and an abstract reference escape semantic environment $e\hat{n}v_r$ mapping an auxiliary function for a transformed version of f to element of $\hat{D}_r$, the global lazy reference escape test function G⊥_refescape? determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of the lazy function f could escape f globally It is defined as follows:

$$\mathsf{G_refescape?}(f,i,j,e\hat{n}v_r) = (\bar{R}_e[\![f''\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\hat{n}v_r[f'' \mapsto \bar{f''}, x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f'' is an auxiliary function for f', f' is a transformed function of f,

$$\bar{f''} = \bar{R}_e[\![f']\!]e\hat{n}v_r,$$

$$y_{ij} = \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, W^{\tau_i}\rangle\rangle,$$

for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{\tau_i}\rangle\rangle,$$

and for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{\tau_i}\rangle\rangle,$$

and τ_i is the type of the i^{th} parameter of f. Then, from the result of the global lazy reference escape test function, we can conclude the following:

- If $\mathsf{G_refescape?}(f,i,j,e\hat{n}v_r) = 0$ then we can conclude that in any possible application of a lazy function f to n arguments, the reference associated with the j^{th} occurrence of the i^{th} delayed argument *does not* escape the function f.
- If $\mathsf{G_refescape?}(f,i,j,e\hat{n}v_r) = 1$ then it means that in some possible application of a lazy function f to n arguments, the reference associated with the j^{th} occurrence of the i^{th} delayed argument *could* escape the function f.

Local Lazy Reference Escape Test

Given a lazy function $f\ x_1\ x_2\ \ldots\ x_n\ =\ body_f$ of arity n in an application $f\ e_1\ \ldots\ e_n$, the position (i,j) of an interesting occurrence of a parameter, and an abstract reference escape semantic environment $e\hat{n}v_r$ mapping an auxiliary function for a transformed version of f

and the free identifiers within e_1 through e_n to elements of $\hat{D}_r$, the global lazy reference escape test function L⊥_refescape? determines whether the reference associated with the j^{th} occurrence of the i^{th} parameter of the lazy function f could escape f in the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$\text{L⊥_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) =$$
$$(\bar{R}_e[\![f''\ x_1\ \ldots\ x_{1o(1)}\ \ldots\ x_n\ \ldots\ x_{no(n)}]\!]\ e\hat{n}v_r[f'' \mapsto \bar{f''}, x_{ij} \mapsto y_{ij}])_{(1)(1)}$$

where f'' is an auxiliary function for f', f' is a transformed function of f,

$$\bar{f''} = \bar{R}_e[\![f']\!]e\hat{n}v_r,$$

$$y_{ij} = \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, (\bar{R}_e[\![e_i]\!]e\hat{n}v_r)_{(2)}\rangle\rangle,$$

for all $k \leq o(i)$ and $k \neq j$,

$$y_{ik} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, (\bar{R}_e[\![e_i]\!]e\hat{n}v_r)_{(2)}\rangle\rangle,$$

and for all $1 \leq m \leq o(l)$ and $l \neq i$,

$$y_{lm} = \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, (\bar{R}_e[\![e_l]\!]e\hat{n}v_r)_{(2)}\rangle\rangle.$$

Then, from the result of the local lazy reference escape test function, we can conclude the following:

- If $\text{L⊥_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = 0$ then we conclude that the reference associated with the j^{th} occurrence of the i^{th} delayed argument *does not* escape f in $(f\ e_1\ \ldots\ e_n)$.
- If $\text{L⊥_refescape?}(f, i, j, e_1, \ldots, e_n, e\hat{n}v_r) = 1$ then it means that the reference associated with the j^{th} occurrence of the i^{th} delayed argument *could* escape f in $(f\ e_1\ \ldots\ e_n)$.

7.2.4 Examples

As an example, consider the following non-strict (with lazy evaluation) program:

```
letrec  f x y = x+y;
        g a = if (a=0) then (f a) else (f 3);
        h c d  = g (c+d);
in   ...
```

where `f, g, h` : $int \rightarrow int \rightarrow int$. The transformed functions `f'`, `g'` and `h'` for `f`, `g` and `h`, based on the program transformation function **N**, are defined as follows:

```
letrec  f' x y = (x []) + (y []);
        g' a = if ((a []) = 0) then (f' lambda().(a []))
               else (f' lambda().3);
        h' c d = g' lambda().((c []) + (d []));
in   ...
```

where f', g', h' : $(* \rightarrow int) \rightarrow (* \rightarrow int) \rightarrow (* \rightarrow int)$ and * denotes the type of a void identifier and a dummy expression. Note that the occurrence of g's parameter a in the expression (f' lambda().(a [])) has an evaluated status.

Lazy Escape Information

The definitions of the abstract lazy escape semantic values f', g', and h' of f', g', and h' are expressed as follows:

$$\begin{aligned} f' &= \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\langle 0, err\rangle\rangle\rangle \\ g' &= \langle 0, \lambda a.(f'_{(2)}\ \langle 0, \lambda().a\ \langle 0, err\rangle\rangle \sqcup (f'_{(2)}\ \langle 0, \lambda().\langle 0, err\rangle\rangle)\rangle \\ &= \langle 0, \lambda a.(f'_{(2)}\ \langle 0, \lambda().a\ \langle 0, err\rangle\rangle)\rangle \\ h' &= \langle 0, \lambda c.\langle c_{(1)}, \lambda d.g'_{(2)}\ \langle (c_{(1)} \sqcup d_{(1)}), \lambda().\langle 0, err\rangle\rangle\rangle\rangle \end{aligned}$$

Let $e\hat{n}v_o = [\texttt{f'} \mapsto f', \texttt{g'} \mapsto g', \texttt{h'} \mapsto h']$.

To find the global escape property of the lazy function h, we apply the lazy global escape test function G_l_escape? as follows:

$$\begin{aligned} \texttt{G_l_escape?}(\texttt{h}, 1, e\hat{n}v_o) &= (\bar{O}_o[\![\texttt{h' c d}]\!]e\hat{n}v'_o)_{(1)} \\ &= 0 \end{aligned}$$

where

$$e\hat{n}v'_o = e\hat{n}v_o[\texttt{c} \mapsto \langle 1, \lambda().\langle 0, W^{(*\rightarrow int)\rightarrow(*\rightarrow int)}\rangle\rangle, \texttt{d} \mapsto \langle 0, \lambda().\langle 0, W^{(*\rightarrow int)\rightarrow(*\rightarrow int)}\rangle\rangle],$$

$$W^{(*\rightarrow int)\rightarrow(*\rightarrow int)} = \lambda x.\langle x_{(1)}, \lambda y.x_{(1)} \sqcup y_{(1)}, err\rangle\rangle.$$

Similarly,

$$\begin{aligned} \texttt{G_l_escape?}(\texttt{h}, 2, e\hat{n}v_o) &= (\bar{O}_o[\![\texttt{h' c d}]\!]e\hat{n}v'_o)_{(1)} \\ &= 0 \end{aligned}$$

where

$$e\hat{n}v'_o = e\hat{n}v_o[\texttt{c} \mapsto \langle 0, \lambda().\langle 0, W^{(*\rightarrow int)\rightarrow(*\rightarrow int)}\rangle\rangle, \texttt{d} \mapsto \langle 1, \lambda().\langle 0, W^{(*\rightarrow int)\rightarrow(*\rightarrow int)}\rangle\rangle],$$

$$W^{(*\rightarrow int)\rightarrow(*\rightarrow int)} = \lambda x.\langle x_{(1)}, \lambda y.x_{(1)} \sqcup y_{(1)}, err\rangle\rangle.$$

Thus, we can conclude that neither the first parameter `c` nor the second parameter `d` of the function `h` escapes `h`.

Lazy Reference Escape Information

The definitions of reference escape semantic values f', g', and h' of `f'`, `g'`, and `h'` are expressed as follows:

$$\begin{aligned}
f' &= \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
g' &= \langle\langle 0,0\rangle, \lambda a.(f'_{(2)}\ \langle\langle 0,0\rangle, \lambda().a\ \langle\langle 0,0\rangle, err\rangle\rangle \sqcup (f'_{(2)}\ \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, err\rangle\rangle\rangle\rangle \\
&= \langle\langle 0,0\rangle, \lambda a.(f'_{(2)}\ \langle a_{(1)}, \lambda().a\ \langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
h' &= \langle\langle 0,0\rangle, \lambda c.\langle c_{(1)}, \lambda d.g'_{(2)}\ \langle (c_{(1)} \sqcup d_{(1)}), \lambda().\langle\langle 0,0\rangle, err\rangle\rangle\rangle\rangle
\end{aligned}$$

The auxiliary functions `f''`, `g''` and `h''` for the transformed functions `f'`, `g'` and `h'` are defined as follows:

```
f'' x y = (x []) + (y []);
g' a1 a2 = if ((a1 []) = 0) then (f' lambda().(a2 []))
               else (f' lambda().3);
h' c d = g' lambda().((c []) + (d []));
```

Then, the definitions of reference escape semantic values f'', g'', and h'' of `f''`, `g''`, and `h''` are expressed as follows:

$$\begin{aligned}
f'' &= \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
g'' &= \langle\langle 0,0\rangle, \lambda a1.\langle a1_{(1)}, \lambda a2.(f'_{(2)}\ \langle a2_{(1)}, \lambda().a2_{(2)}\ \langle\langle 0,0\rangle, err\rangle\rangle\rangle \\
h'' &= \langle\langle 0,0\rangle, \lambda c.\langle c_{(1)}, \lambda d.g'_{(2)}\ \langle (c_{(1)} \sqcup d_{(1)}), \lambda().\langle\langle 0,0\rangle, err\rangle\rangle\rangle\rangle
\end{aligned}$$

Let $e\hat{n}v_r = [\texttt{f'} \mapsto f', \texttt{f''} \mapsto f'', \texttt{g'} \mapsto g', \texttt{g''} \mapsto g'', \texttt{h'} \mapsto h', \texttt{h''} \mapsto h'']$.

To find the global reference escape property of the non-strict function `h`, we apply the non-strict global reference escape test function G⊥_refescape? as follows:

$$\begin{aligned}
\mathsf{G_\bot_refescape?}(\texttt{h}, 1, 1, e\hat{n}v_r) &= (\bar{R}_e[\![\texttt{h'' c d}]\!]e\hat{n}v'_r)_{(1)(1)} \\
&= 0
\end{aligned}$$

where

$$\begin{aligned}
e\hat{n}v'_r = e\hat{n}v_r[&\texttt{c} \mapsto \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, W^{(*\to int)\to(*\to int)}\rangle\rangle, \\
&\texttt{d} \mapsto \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{(*\to int)\to(*\to int)}\rangle\rangle]
\end{aligned}$$

Similarly,

$$\begin{aligned} \mathsf{G_l_refescape?}(\mathtt{h}, 2, 1, e\hat{n}v_r) &= (\bar{R}_e[\![\mathtt{h'' \ c \ d}]\!]e\hat{n}v'_r)_{(1)(1)} \\ &= 0 \end{aligned}$$

where

$$\begin{aligned} e\hat{n}v'_r = e\hat{n}v_r[\mathtt{c} \mapsto \langle\langle 0,0\rangle, \lambda().\langle\langle 0,0\rangle, W^{(*\rightarrow int)\rightarrow(*\rightarrow int)}\rangle\rangle, \\ \mathtt{d} \mapsto \langle\langle 1,0\rangle, \lambda().\langle\langle 1,0\rangle, W^{(*\rightarrow int)\rightarrow(*\rightarrow int)}\rangle\rangle] \end{aligned}$$

Thus, we can conclude that neither the reference associated with the first occurrence of the first parameter `c` of the function nor the reference associated with the first occurrence of the second parameter `d` of the function `h` escapes `h`.

7.3 Escapement under Evaluation with Strictness

Strictness information about parameters of a lazy function can be used for improving the lazy evaluation strategy by directly evaluating strict arguments before they are passed to the function body. This evaluation strategy can be considered as a combination of applicative-order evaluation (strict semantics) and lazy evaluation (non-strict semantics). The escapement property under this evaluation strategy can be effected by transforming non-strict programs using the strictness information about non-strict functions. We assume that with individual application nodes in the bodies of functions strictness information is annotated to indicate strict applications ([66]). The new source-to-source transformation function $\mathbf{M}$ of type $Exp \rightarrow \mathsf{Powerset}(Id) \rightarrow Exp$ for non-strict programs with strictness annotations is defined in Figure 7.4.

$$
\begin{array}{lcl}
\mathbf{M}[\![c]\!]\ FId & \Rightarrow & \lambda().c \\
\mathbf{M}[\![x]\!]\ FId & \Rightarrow & \text{if } (x \in FId) \text{ then } x \text{ else } (x\ \texttt{[]}) \\
\mathbf{M}[\![e_1 + e_2]\!]\ FId & \Rightarrow & (\mathbf{M}[\![e_1]\!]\ FId) + (\mathbf{M}[\![e_2]\!]\ FId) \\
 & & \text{/* same for strict operators like } (e_1 - e_2) \text{ and } (e_1 = e_2) \text{ */} \\
\mathbf{M}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\ FId & \Rightarrow & \texttt{if}\ (\mathbf{M}[\![e_1]\!]\ FId) \\
 & & \texttt{then}\ (\mathbf{M}[\![e_2]\!]\ FId) \\
 & & \texttt{else}\ (\mathbf{M}[\![e_3]\!]\ FId) \\
\mathbf{M}[\![e_1 e_2]\!]\ FId & \Rightarrow & \text{if } e_1 \in \{\ \texttt{car, cdr}\ \} \text{ /* strict operator */} \\
 & & \text{then } (\mathbf{M}[\![e_1]\!]\ FId)\ \ (\mathbf{M}[\![e_2]\!]\ FId) \\
 & & \text{elseif } (\mathit{strict_application}?) \text{ /* strict application */} \\
 & & \text{then } (\mathbf{M}[\![e_1]\!]\ FId)\ \ (\mathbf{M}[\![e_2]\!]\ FId) \\
 & & \text{else /* non-strict application */} \\
 & & (\mathbf{M}[\![e_1]\!]\ FId)\ \ (\lambda().\mathbf{M}[\![e_2]\!]\ FId) \\
\mathbf{M}[\![\texttt{lambda}(x).e]\!]\ FId & \Rightarrow & \texttt{lambda}(x).(\mathbf{M}[\![e]\!]\ FId) \\
\mathbf{M}[\![\texttt{letrec}\ x_1 = e_1; \ldots x_n = e_n; \texttt{in}\ e]\!]\ FId & \Rightarrow & \texttt{letrec} \\
 & & x_1 = \mathbf{M}[\![e_1]\!]\ (FId \cup \{x_1, \ldots, x_n\}); \\
 & & \ldots \\
 & & x_n = \mathbf{M}[\![e_n]\!]\ (FId \cup \{x_1, \ldots, x_n\}); \\
 & & \texttt{in} \\
 & & \mathbf{M}[\![e_1]\!]\ (FId \cup \{x_1, \ldots, x_n\}) \\
\mathbf{M}_P[\![pr]\!] & \Rightarrow & \mathbf{M}[\![pr]\!]\ \emptyset
\end{array}
$$

Figure 7.4: Non-strict Program Transformation with Strictness Information

Chapter 8

Storage Management Optimizations

The escape information that is inferred at compile-time from the semantic analyses which have been described in preceding chapters, including escape analysis, refined escape analysis, reference escape analysis, and order-of-demand analysis, allows a variety of storage management optimizations in functional language implementations.

In this chapter, we propose a variety of optimization techniques, based on statically inferred escape information, to reduce the storage management overhead in functional language implementation, including stack allocation, explicit reclamation, in-place reuse of garbage cells, reference counting elimination, block allocation/reclamation, and improving generational garbage collection.

8.1 Stack Allocation

The need for heap allocation of objects, such as arguments and locally defined objects within a higher-order function, arises because they may outlive a call to that function, that is, the environment in which they were created. Thus, they cannot be deallocated when the environment is left and its bindings are deallocated. For example, when a partial application or explicit lambda expression is evaluated to produce a closure, the storage must be allocated and the values must be moved into the closure. Generally, closures (the code pointer and environment) need to be stored in the heap at run-time and reclaimed by some garbage collection process.

On most current systems, the stack allocation and deallocation is usually more efficient

both in space and in time than the heap allocation and garbage-collected reclamation, although the overhead of a garbage collection can be quite small with a sufficiently large amount of physical memory and a sophisticated garbage collection strategy [5]. The motivation is to avoid allocation of objects in the heap and instead to allocate them in the stack. Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in stack-like storage and cheaply reclaimed without invoking garbage collection. Optimizations that convert heap allocations into stack-like storage allocations are expected to save significant time. Converting heap allocation to stack allocation may also permit further optimizations.

Escape information about parameters of a strict function that is inferred through the (refined) escape analysis can be used for the stack allocation optimization. Consider *any* expression of the form of

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a function with n parameters, and each e_i, $1 \leq i \leq n$, is *any* possible expression. Let e_i be an expression whose evaluation requires to storage allocation. The storage for the value of e_i generally needs to be allocated from a heap, because the lifetime of e_i cannot be predicted. Based on the *global* escape information about the parameters of f, more efficient storage allocation can safely be done for e_i as follows:

- If the i^{th} parameter of f does *not* escape f globally then allocate the storage for e in the *stack* where the activation record for f resides, else allocate the storage for e in the *heap*.
- If the value of e_i is a constant list and the top j spines of the i^{th} parameter of f does *not* escape f globally then allocate the storage for cells at the top j spines in the *stack* where the activation record for f resides, and allocate the storage for cells at the remaining spines in the *heap*.

Consider a *particular* subexpression of

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a function with n parameters. Similarly, the *local* escape information of parameters of f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ can be used for more efficient storage allocation for e_i as follows:

- If the i^{th} parameter of f does *not* escape f in the particular application ($f\ e_1\ \ldots\ e_i\ \ldots\ e_n$) then allocate the storage for e_i in the *stack* where the activation record for f resides, else allocate the storage for e_i in the *heap*.

- If the value of e_i is a constant list and the top j spines of the i^{th} parameter of f does *not* escape f in ($f\ e_1\ \ldots\ e_i\ \ldots\ e_n$) then allocate the storage for cells at the top j spines in the *stack* where the activation record for f resides, and allocate the storage for cells at the remaining spines in the *heap*.

As an example, consider the program given in Chapter 2 as follows:

```
letrec  g a b = if (a < b) then 0 else a;

        h c d = if (c < d) then d else 0;

        map f l = if (null l) then nil
                  else cons (f (car l)) (map f (cdr l));

in  ... (map (g 3) [1,3,5]) ... (map (h 3) [1,3,5]) ...
```

From the global escape analysis, we know that the first parameter `f` of the function `map` can never escape `map` globally. Thus, in any subexpression of (`map` e_1 e_2) where the evaluation of e_1 requires storage allocation, that storage for e_1 can safely be allocated in the stack in which the activation record for `map` resides. So, the storage for the closures representing `(g 3)` and `(h 3)` can be allocated in the stack instead of the heap.

Even though we cannot conclude from the global escape analysis that the second parameter `l` of the function `map` does not escape `map` globally, the local escape analysis tells us that the second parameter `l` of the function `map` does not escape `map` locally in (`map (g 3) [1,3,5]`), while the second parameter `l` of the function `map` does escape `map` locally in (`map (h 3) [1,3,5]`). Thus, the storage for the cons cells of `[1,3,5]` in (`map (g 3) [1,3,5]`) can also be allocated in the stack instead of allocating in the heap.

The global refined escape analysis tells us more refined escape information about the second parameter `l` of `map`, i.e. the top spine of `l` does not escape `map` globally. Thus, the storage for the cons cells of `[1,3,5]` in (`map (h 3) [1,3,5]`) can also be allocated in the stack instead of the heap.

Escape information about the parameters of a non-strict (lazy) function that is inferred through the escape analysis can be used for the stack allocation optimization. Consider *any*

expression of the form of

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a lazy function with n parameters, and each e_i, $1 \leq i \leq n$, is *any* possible expression. Each argument e_i to f requires storage allocation for representing its delayed evaluation such as a *thunk* in the normal-order evaluation model or a *self-modifying thunk* in the lazy evaluation model. Generally, the storage for each e_i needs to be allocated in a heap. Based on the *global* escape information about the parameters of f, more efficient storage allocation can safely be performed for the thunk of each e_i as follows:

- If the i^{th} parameter of f does *not* escape f globally then allocate the storage for the thunk of e_i in the *stack* where the activation record for f resides, else allocate the storage for the thunk of e_i in the *heap*.

Similarly, the *local* escape information about the parameters of f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ can be used for more efficient storage allocation for the thunk of each e_i as follows:

- If the i^{th} parameter of f does *not* escape f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ then allocate the storage for the thunk of e_i in the *stack* where the activation record for f resides, else allocate the storage for the thunk of e_i in the *heap*.

As an example, consider the following lazy program from Chapter 7:

```
letrec  f x y = x+y;
        g a = if (a=0) then (f a) else (f 3);
        h c d  = g (c+d);
in  ...  h (f 1 2) (f 3 4) ...
```

From the global escape analysis, we know that neither the first parameter `c` nor the second parameter `d` of the function `h` escapes `h`. Thus, in any subexpression of (`h` e_1 e_2), the storage for the thunks for e_1 and e_2 can safely be allocated in the stack where the activation record `map` resides. So, the self-modifying thunks for both `(f 1 2)` and `(f 3 4)` in `h (f 1 2) (f 3 4)` could be allocated in the stack where the activation record for `h` resides.

Safety Issue

From a practical point of view, however, there are safety issues in the application of storage allocation optimization, i.e. unrestricted application of the stack allocation optimization may be unsafe in the sense that it can convert programs that run well into programs that

fail (due to stack overflow). The safety condition is that the stack allocation optimization should not convert a program that runs robustly into one that does not ([20], [21]).

8.2 Explicit Reclamation

When heap-allocated objects are no longer needed, they can often be reclaimed into a free storage list explicitly by the program without invoking the garbage collection. This is done by identifying at compile time where run-time storage management decisions can be made, i.e. the places in a program where storage can safely be collected. When we can predict at compile time when cells will become garbage, we can embed into the executable program additional operations which immediately link the garbage cells to the available free list. Thus, we can avoid some of the expensive operations usually used to detect garbage at run time.

Detecting Sharing using Escape Information

Sharing information about cells can be determined using the escape information.

Theorem 8.1 (Sharing Information) *Let f be a function which takes n arguments such that d_i is the number of spines of the i^{th} parameter of f for $i = 1 \ldots n$, and let f return a list with d_f spines. If esc_i be the number of escaping spines of the i^{th} parameter of f for $i = 1 \ldots n$ (statically inferred by escape analysis), then*

1. *all cells in the top $(d_f - max\{min\{esc_1, (d_1 - u_1)\}, \ldots, min\{esc_n, (d_n - u_n)\}\})$ spines of the result of $(f\ e_1 \ldots e_n)$ are unshared for arguments $e_1, \ldots . e_n$ such that u_i is the number of unshared spines of e_i.*
2. *all cells in the top $(d_f - max\{esc_1, \ldots, esc_n\})$ spines of the result of $(f\ e_1 \ldots e_n)$, for any set of arguments $e_1, \ldots . e_n$, are unshared.*

Proof : 1. The number of spines of e_i that are shared is $(d_i - u_i)$. The number of shared spines of e_i that could escape f is $min\{esc_i, (d_i - u_i)\}$. In the result of $(f\ e_1 \ldots e_n)$, the bottom $max\{min\{esc_i, (d_i - u_i)\}\}$ spines will be shared. Thus, all cells in the top $(d_f - max\{min\{esc_1, (d_1 - u_1)\}, \ldots, min\{esc_n, (d_n - u_n)\}\})$ spines of the result are not shared.

2. Since we consider any set of arguments $e_1, \ldots . e_n$, and we have no sharing information of e_i, we assume that $u_i = 0$ as the worst-case. Then, $min\{esc_i, (d_i - 0)\} = esc_i$ because

$esc_i \leq d_i$. Thus, all cells at top $(d_f - max\{min\{esc_1, \ldots, esc_n\}\})$ spines of the result are not shared. □

As an example, consider the program given in Chapter 3 as follows:

```
letrec ps x = if (null x) then nil
              else letrec y = split (car x) (cdr x) nil nil;
                   in append (ps (car y))
                             (cons (car x) (ps (car (cdr y))));

       split p x l h = if (null x) then (cons l (cons h nil))
                       elseif (car x)<p then
                              split p (cdr x) (cons (car x) l) h
                       else split p (cdr x) l (cons (car x) h);

       append x y = if (null x) then y
                    else cons (car x) (append (cdr x) y);
in ps [5,2,7,1,3,4]
```

The function `ps` takes a list with one spine as its argument, and returns a list with one spine. From the global refined escape analysis, we know that no spine of the argument escapes `ps` globally. Thus, for any expression (`ps` e) where e is a list with one spine, we conclude that the top spine of the result list of (`ps` e) is *not* shared. The function split takes four arguments `p, x, l` and `h` whose type are *int*, *int list*, *int list* and *int list*, respectively, and returns a list with two spines. From the global refined escape analysis, we know that none of the first parameter `p`, all but the top spine of the second parameter `x`, and all of the third and fourth parameters `l` and `h` escape `split` globally. Thus, we conclude that, for any subexpression of (`split` e_1 e_2 e_3 e_4) where each e_i is any possible expression, the top spine of the result list of (`split` e_1 e_2 e_3 e_4) is *not* shared.

We can detect some garbage objects at compile-time using sharing and escape information.

Theorem 8.2 *Consider an expression* $(f\ e_1, \ldots, e_i, \ldots, e_n)$*, where* f *is a function of arity* n*. Then,*

1. *If* e_i *is a function type and the* i^{th} *parameter of* f *does not escape* f*, then the storage for representing the closure of* e_i *is garbage after the execution of the expression, and thus can safely be reclaimed after the execution of the expression.*

2. *If e_i is a list with top u_i unshared spines and the bottom esc_i spines of the i^{th} parameter with d_i spines escape f, then all cells at the top $min\{u_i, (d_i - esc_i)\}$ spines of e_i are garbage after the execution of the expression, and thus can safely be reclaimed after the execution of the expression.*

Proof : Since the number of unshared spines of e_i is u_i, all cells in the top u_i spines of e_i are unshared before the execution of f. Since the bottom esc_i spines of e_i escape f, the top $(d_i - esc_i)$ spines of e_i do not escape f. The spines of e_i that do not escape f and are not shared will be inaccessible at the end of execution of f. Thus, the top $min\{u_i, (d_i - esc_i)\}$ spines of e_i become garbage during the execution of the body of f and are garbage after the execution of f. □

Escape information that is inferred from the (refined) escape analysis can be used for the explicit reclamation optimization. Consider any subexpression of the form of

$$(f\ e_1 \ldots e_{i-1}\ (g\ e'_1 \ldots e'_m)\ e_{i+1} \ldots e_n)$$

where f and g are functions with arity n and m, respectively. Let the result of $(g\ e'_1 \ldots e'_m)$ be a heap-allocated object. Generally, that object is reclaimed by garbage collection. Using the *global* escape information about the parameters of f, more efficient storage reclamation can safely be done for the result of $(g\ e'_1 \ldots e'_m)$ as follows:

- When the i^{th} parameter of f is a function type : If the i^{th} parameter of f does not escape f globally then the subexpression can safely be transformed into

  $$\text{RECLAIM}_i(f\ e_1 \ldots e_{i-1}\ (g\ e'_1 \ldots e'_m)\ e_{i+1} \ldots e_n)$$

  where RECLAIM_i reclaims the i^{th} argument of f.

- When the i^{th} parameter of f is a list type with d_i spines : If the bottom esc_i spines of the i^{th} parameter of f escapes f then the subexpression can safely be transformed into

  $$\text{RECLAIM}_i^s(f\ e_1 \ldots e_{i-1}\ (g\ e'_1 \ldots e'_m)\ e_{i+1} \ldots e_n)$$

  where

  $$s = min\{(d_i - max\{esc'_1, \ldots, esc'_m\}), (d_i - esc_i)\},$$

  esc'_i, $1 \leq i \leq m$, is the number of bottom spines of the i^{th} parameter of g that escape g, and RECLAIM_i^s reclaims all cells in the the top s spines of the i^{th} argument of f.

As an example, consider the same program as in section 8.1. From the global refined escape analysis, we know that `append` returns all of its second argument `y`, and all but the top spine of the first argument `x`. We also know that, for any expression (`ps` e) where e is a list with one spine, the top spine of the result of (`ps` e) is *unshared*. Thus, the definition of `ps` can be transformed into `PS` as follows:

```
PS x = if (null x) then nil
       else letrec y = split (car x) (cdr x) nil nil;
            in RECLAIM<1,1> append (PS (car y))
               (cons (car x) (PS (car (cdr y))));
```

where `RECLAIM<1,1>` reclaims all cells in the top spine of the first argument of `append`. Note that, from the global escape analysis, we know that the top spine of `ps`'s parameter does not escape from `ps`, only some elements do. So, each cell of the top spine of the list `ps [5,2,7,1,3,4]` could be allocated in the activation record for `ps`.

8.3 In-place Reuse

When heap-allocated objects are no longer needed and other objects need to be allocated, the storage can be reused directly by the program without new allocation and without invoking garbage collection. The motivation is to replace the allocation of new cells by direct reuse of previously deallocated garbage cells.

Theorem 8.3 *Consider an expression* $(f\ e_1, \ldots, e_i, \ldots, e_n)$*, where* f *is a function with* n *parameters. Then, if each* e_i *is a list with its top* u_i *spines unshared, and the bottom* esc_i *spines of the* d_i *spines of the* i^{th} *parameter of* f *escape* f *then all cells in the top* $min\{u_i, (d_i - esc_i)\}$ *spines of* e_i *are garbage after the execution of the expression, and thus can safely be reused during the execution of the expression.*

Proof : Since the number of unshared spines of e_i is u_i, all cells at top u_i spines of e_i are unshared before the execution of f. Since the bottom esc_i spines of e_i escape f, the top $(d_i - esc_i)$ spines of e_i do not escape f. The spines of e_i that do not escape f and are not shared will be inaccessible at the end of the execution of f. Thus, the top $min\{u_i, (d_i - esc_i)\}$ spines of e_i become garbage during the execution of the body of f and are garbage after the execution of f. □

Escape information about parameters of a strict function that is inferred by the (refined) escape analysis can be used for the in-place reuse optimization. Consider an expression of the form of

$$(f\ e_1 \ldots\ e_i\ \ldots e_n)$$

where f is a function with n parameters, the i^{th} parameter of f is a list type with d_i spines, and there occurs some `cons` in the body of f. Let all cons cells at the top u_i spines of the result of e_i be unshared. Generally, each `cons` appearing in the body of f allocates a new cons cell in the heap, and such cons cells are reclaimed by garbage collection. Using the *global* escape information of parameters of f, more efficient storage allocation and reclamation can safely be performed as follows:

- If the bottom esc_i spines of the i^{th} parameter of f escapes f globally then the subexpression can safely be transformed into

  $$(f_i^s\ e_1 \ldots\ e_i\ \ldots e_n)$$

  where

  $$s = min\{u_i, (d_i - esc_i)\},$$

  and f_i^s is a new version of f which directly reuses cells in the top s spines of the i^{th} argument of f for new cells needed in the body of f.

Consider a function f as follows:

$$f\ x_1\ \ldots\ x_n\ =\ \ \ldots\ (\texttt{cons}\ e_1\ e_2)\ \ldots$$

- If there is no use of the i^{th} parameter x_i of f after the evaluation of the subexpression $(\texttt{cons}\ e_1\ e_2)$ then a new version f_i^s of f which uses the in-place reuse optimization can be defined as follows:

  $$f_i^s\ x_1\ \ldots\ x_n\ =\ \ \ldots\ (\texttt{DCONS}\ x_i\ e_1\ e_2)\ \ldots$$

  where `DCONS` is a destructive version of `cons` defined by

  $$\begin{aligned}\texttt{DCONS}\ a\ b\ c\ \ =\ \ & \{p := a;\\ & \texttt{car}(a) := b;\\ & \texttt{cdr}(a) := c;\\ & return(p)\}.\end{aligned}$$

As an example, consider the partition sort program from the previous section. From the global refined escape analysis, we know that `append` returns all of its second argument `y`, and all but the top spine of the first argument `x`. The top spine of the list that `ps` returns is

unshared. We also know that, for any expression (ps e) where e is a list with one spine, the top spine of the result of (ps e) is *unshared*. Thus, the definition of ps can be transformed into PS as follows:

```
PS x = if (null x) then nil
       else letrec y = split (car x) (cdr x) nil nil;
            in APPEND (PS (car y))
               (cons (car x) (PS (car (cdr y))));
```

where APPEND is a version of append in which cells are directly reused. It is defined by

```
APPEND x y = if (null x) then y
             else DCONS x (car x) (APPEND (cdr x) y);
```

Furthermore, if we know that the top spine of the argument of ps is unshared, then the definition of ps can be transformed into PS' as follows:

```
PS' x = if (null x) then nil
       else letrec y = split (car x) (cdr x) nil nil;
            in APPEND (PS' (car y))
               (DCONS x (car x) (PS' (car (cdr y))));
```

8.4 Reference Counting Elimination

Reference counting is a storage reclamation method in which each object contains a count, called the reference count, of the number of references (pointers) pointing to it. When an object is first allocated, its reference count is set to one. The reference count is updated during execution as follows: Each time a new reference to an object is created, the object's reference count is incremented by one. Each time a reference to an object is destroyed, the object's reference count is decremented by one. When an object's reference count becomes zero, it can be reclaimed and the reference count of each object that it points to is decremented. Though the reference counting strategy has disadvantages, such as storage fragmentation and the inability to reclaim cyclic structures, its major advantage is that storage reclamation occurs incrementally throughout program execution; storage can be reclaimed as soon as it has become garbage. It is also especially suitable in multiprocessor architectures with distributed memory, since reference counting is an inherently real-time and localized activity.

The major overheads that are incurred in reference counting schemes are as follows:

- Space overhead for maintaining a reference count in each object.
- Time and code overhead for updating reference counts when references are created or destroyed.
- Communication overhead for manipulating a remote reference and for synchronizing the operations on reference counts in distributed memory environments ([57]).

We describe a method for reducing the time, code, and communication overhead of reference counting in both uniprocessor and multiprocessor environments by compile-time program analysis. Our approach is based on the observation that such overheads can be reduced by avoiding unnecessary reference count updates using statically inferred information about the *lifetime* of each reference. The lifetime of a reference to an object is the period from when the reference is created until its last use. Suppose O is an object that is active at some time t_0 during execution. Since this object is active, there is at least one reference pointing to it. If its reference count O_{rc} is n then there are exactly n references pointing to it. Suppose A is one of those references. Now, suppose that a new reference B to the object is created. The current reference count of the object is incremented, i.e. $O_{rc} := O_{rc} + 1$. At some later time t_1, suppose that B is discarded. Then, the current reference count of the object is decremented, i.e. $O_{rc} := O_{rc} - 1$.

If we can determine, at compile-time, that A will still be active at time t_1, then no reference count operations are required when B is created or destroyed. Since the reference count of O always remains greater than or equal to one from time t_0 to time t_1, O will *not* to be reclaimed between time t_0 and time t_1. Thus, the reference count updating operation for the reference B can be avoided. This avoidance optimization is safe because any object which is still active will not be reclaimed.

Reference escape information that is inferred from the reference escape analysis can be used for the reference counting elimination optimization. Let f be a strict or lazy function defined as follows:

$$f\ x_1\ \ldots\ x_n\ =\ \ldots\ x_{ij}\ \ldots$$

where x_{ij} is the j^{th} occurrence of the i^{th} parameter x_i in the body of f such that it causes the creation of a reference to the i^{th} argument of f during the evaluation of an application of f to n arguments. In classic reference counting scheme, the reference count of the i^{th} argument of f needs to be updated when the reference associated with x_{ij} is created and

destroyed. Based on the *global* reference escape information of occurrences of the parameters of f, more efficient reference count updating can safely be done as follows:

- If x_{ij} *does not* reference-escape f globally then just create and destroy the reference without reference count updates else create and destroy the reference with reference count updates

Furthermore, given two references A and B to a heap allocated object, the relative lifetimes of A and B can be computed by determining if there is a scope from which one of them escapes but not the other. If so, when the shorter-lived reference is created and destroyed, no reference count operations are necessary. In some programs, in fact, our analysis can determine if some reference R to an object outlives all others. Thus, the object can be reclaimed as soon as R is destroyed. No other reference counting operations are needed. Notice, however, that it may not be possible to determine lifetime of R (if it is embedded in a structure, for instance), and thus of the object, at compile time.

As an example, consider the program given in Chapter 4:

```
letrec  map f l = if (null l) then nil
                  else cons (f (car l)) (map f (cdr l));

        sum l = if (null l) then 0
                else (car l) + sum (cdr l);

        addsum x y = cons x (cons y (cons
                     (map (lambda(z). (sum Y) + z) X) nil));
in ...
```

From the reference escape analysis, we know that the reference associated with the second occurrence of each parameter `x` and `y` of `addsum` does not escape. Thus, updates on the reference count of each parameter could be avoided when creating and deleting the reference. Consider the following lazy program from Chapter 7:

```
letrec  f x y = x+y;
        g a = if (a=0) then (f a) else (f 3);
        h c d  = g (c+d);
in   ...
```

From the reference escape analysis, we know that neither the reference associated with the first occurrence of the first parameter `c` of the function nor the reference associated with

the first occurrence of the second parameter `d` of the function `h` escapes `h`. Thus, reference count updatings can be avoided when these references are created and deleted.

8.5 Block Allocation/Reclamation

In this scheme, a number of objects are allocated together in a contiguous block of a heap storage and the whole block is put on the free list, rather than the individual objects. This allows reclamation of larger segments of storage, and reduces run-time overhead by avoiding the traversal of the individual objects (in mark-sweep collection, for instance).

Escape information about the parameters of a strict function that is inferred at compile-time through the (refined) escape analysis can be used for the block allocation/reclamation optimization. Consider *any* subexpression of the form of

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a function with n parameters, and each e_i, $1 \leq i \leq n$, is *any* possible expression. Let e_i be an expression whose evaluation requires the allocation of a number of storage cells. Generally, each individual cell of the value of e_i needs to be allocated in a different area from a heap. Based on the *global* escape information of parameters of f, more efficient storage allocation can safely be performed for e_i as follows:

- If the top j spines of the i^{th} parameter of f does *not* escape f globally then allocate all cells in the top j spines together in the same block in the heap

Consider a *particular* expression

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a function with n parameters. Similarly, the *local* escape information about the parameters of f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ can be used for more efficient storage allocation for e_i as follows:

- If the top j spines of the i^{th} parameter of f does *not* escape f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ then allocate all cells at the top j spines together in the same block in the heap

As an example, consider the program in section 8.1. From the escape analysis, we know that the top spine of `ps`'s parameter does not escape from `ps`, only some elements do. Thus, each cells of the top spine of the list `ps [5,2,7,1,3,4]` could be allocated in the same block in the heap.

8.6 Improving Generational Garbage Collection

Generational garbage collection groups objects into areas according to their predicted lifetimes, and collect areas independently and asynchronously.

Escape information about parameters of a strict function that is inferred at compile-time through the (refined) escape analysis can be used for improving the generational garbage collection schemes. Consider *any* expression of the form of

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a function with n parameters, and each e_i, $1 \leq i \leq n$, is *any* possible expression. Let e_i be an expression whose evaluation requires storage for the result. In classic generational garbage collection, the storage for the value of e_i needs to be allocated in a new area (the youngest generation) in a heap. Based on the *global* escape information about the parameters of f, more efficient selection of generations can safely be performed for e_i as follows:

- If the i^{th} parameter of f does *not* escape f globally then allocate the storage for e_i in a region with current youngest generation else allocate the storage for e_i in some region with older generations.

- If the value of e_i is a list and the top j spines of the i^{th} parameter of f does *not* escape f globally then allocate the storage for cells at the top j spines in a region with the youngest generation and allocate the storage for cells in the remaining spines in some region with older generations.

Consider a *particular* expression

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a function with n parameters. The *local* escape information about the parameters of f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ can be used for more efficient storage allocation for e_i as follows:

- If the i^{th} parameter of f does *not* escape f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ then allocate the storage for e_i in a region with the youngest generation else allocate the storage for e_i in some region with older generations.

- If the value of e_i is a constant list and the top j spines of the i^{th} parameter of f does *not* escape f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ then allocate the storage for cells at the top j spines in a region with the youngest generation and allocate the storage for cells in the remaining spines in some region with older generations.

Escape information about the parameters of a non-strict function that is inferred at compile-time through the escape analysis can be used for improving the generational garbage collection scheme. Consider *any* expression of the form

$$(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$$

where f is a lazy function with n parameters, and e_i, $1 \leq i \leq n$, is *any* possible expression. Each argument e_i to f requires the allocation of storage for representing its delayed evaluation, such as a thunk in the normal-order evaluation model or a self-modifying thunk in the lazy evaluation model. Generally, the storage for each e_i needs to be allocated from a heap. Based on the *global* escape information about the parameters of f, more efficient selection of the generations of storage for e_i can be done as follows:

- If the i^{th} parameter of f does *not* escape f globally then allocate the storage for the thunk for e_i in a region with the current youngest generation else allocate in some region with older generations.

Similarly, the *local* escape information about the parameters of f in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ can be used for more efficient storage allocation for the thunk for each e_i in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ as follows:

- If the i^{th} parameter of f does *not* escape f locally in $(f\ e_1\ \ldots\ e_i\ \ldots\ e_n)$ then allocate the storage for thunk of e_i in a region with the current youngest generation else allocate in some region with older generations.

Chapter 9

Related Work, Conclusions, and Future Work

In this chapter, we survey some previous work related this thesis, summarize the contributions of this thesis, and suggest some further research in this area.

9.1 Related Work

Escape and Lifetime Information

Brooks, Gabriel, and Steele [16] described a simple first-order escape analysis for numbers in LISP, but did not extend it for arbitrary objects. Orbit, an optimizing compiler for Scheme, used a simple first-order escape analysis to stack allocate closures [42]. Hudak and Kranz [42] used a simple first-order escape analysis of a non-strict functional language for stack allocation of self-modifying thunks.

There have been a number of papers describing analyses for optimizing storage of lists and other structures. Most of these analyses have been first-order (i.e. not accounting for higher-order functions.) Ruggieri and Murtagh ([71], [72]) described a lifetime analysis for a language with side-effects and complex data structures, but, again, it is first-order.

Inoue, Seki, and Yagi [49] described an analysis for functional languages to detect, and reclaim run-time garbage cells based on formal language theory and grammars. They focused only on the explicit reclamation of cons cells. It is unclear if this approach could be extended for higher-order languages. Besides being higher-order, the escape analysis that we describe in this thesis is a more general lifetime analysis that can be applied to objects other than lists.

Jones and LeMetayer [51] described an algorithm, based on forward and backward analyses, for detecting sharing of objects in first-order functional languages, and describe a method for reusing of cells based on the sharing analysis. We use only forward analysis providing, perhaps, a simpler conceptual framework.

Chase, Wegman, and Zadeck [22] described a first-order analysis for LISP that constructs graphs representing possible list structures and analyzes the graphs for possible storage optimizations. Our analysis, in contrast, benefits from a type system that restricts the ways that lists can be created and that restricts the kinds of sharing that can occur within a list (for example, one cannot write `cons(x,x)`.)

Deutsch [28] presented a lifetime and sharing analysis for higher-order languages. The analysis consisted of defining a low-level operational model for a higher-order functional language, translating a program into a sequence of operations in this model, and then performing an analysis to determine the lifetimes of dynamically created objects. The approach is one of collecting interpretation, in that it analyzes a whole program to infer properties of program points. Our approach is to define a high-level non-standard semantics that in many ways is similar to the standard semantics and captures the precise escape behavior caused by the constructs in a functional language. We then define an abstraction of these semantics which provides less precise information but which allows the analysis to be performed at compile time. The advantage of our analysis lies in its conceptual simplicity and less computational cost (compared to a collecting interpretation).

Deutsch and Bobrow [29] proposed a method for reducing the overhead of updating reference counts in which reference counting activities are *deferred* by being stored into a file called a transaction file instead of being immediately performed. Reference counts are then adjusted at suitable intervals. Barth [9] showed that this particular reference counting scheme could benefit from compile time optimization by generating fewer transactions (reference counting activities) based on compile time analysis of *first-order* programs.

Using the idea of weighted references, i.e. each reference carries a weight such that the sum of the weights of all references to an object is equal to the reference count of the object, there have been a variety of works [10,82], in which, when a new reference is created to an object, no access to the object is needed. Goldberg [32] presented a generation-based approach for distributed systems that also avoids reference count operations when a reference is created, and also described the applicability of escape information among references to reference counting schemes, but did not present the analysis.

In the garbage collection area, using the lifetime of objects(*not statically inferred*),

Lieberman and Hewitt [58] suggested a copying garbage collection in which storage is divided into regions according to ages. Hudak [39] presented a semantic model for describing the number of active pointers to objects for an applicative-order interpreter of a first-order function language, and a variety of its abstractions based on abstract and collecting interpretations.

Strictness, Evaluation Order, and Evaluation Status Information

Information about which arguments of a function will definitely be demanded, called strictness information, is used to optimize lazy evaluation by converting lazy evaluation into applicative-order evaluation and thus reducing the overhead of lazy evaluation. Information about the order of evaluation of the arguments to a function can be useful for a number of optimizations, including copy elimination [11,30,35] and process scheduling in a parallel system [13]. Information on the status of evaluation of arguments when they are demanded is useful for eliminating unnecessary checking [14].

There have been many papers published in the field of strictness analysis for languages with higher-order functions, polymorphism, and non-flat domains ([18], [45], [36], [37], [43], [1], [80], [81]), but these do not provide information about the order or status of evaluation of arguments. Issues of order of evaluation are addressed in [41], and a variety of models for obtaining order of evaluation information in a first-order non-strict functional language without non-flat domains are described in [12]. However, those models are somewhat complex and the extension to higher-order languages is not clear.

Bloss [13] proposed a *path analysis* which provides a range of static information including strictness, evaluation-order, and evaluation-status information in a first-order lazy functional language. The analysis consists of determining the set of all possible evaluation sequences called *paths* and extracting interesting information from the set. Since the size of the abstract semantic domain is very large, the time complexity of the analysis for first-order languages is significantly higher than that of strictness analysis. Path analysis can also be extended to higher-order languages [11]. However, the complexity becomes even worse in the case of higher-order languages, and it is not clear how to extract much useful information from higher order path analysis. In [35] a method, similar to the path model, for extracting some information about order of evaluation in a higher-order strict functional language was presented.

Draghicescu and Purushothaman [30] presented a compositional analysis for obtaining at compile-time some information about the order of evaluation of variables in a first-order

non-strict functional language with both lazy evaluation and other evaluation strategies using strictness information. This approach is based on strictness information and achieves a lower complexity than path analysis by analyzing relations between parameters instead of computing all possible paths of parameters. Our work deals with higher-order functional languages and uses a smaller abstract domain and thus shows much lower complexity.

Polymorphic Invariance

Abramsky [1] extended the strictness analysis for monomorphic languages to polymorphic languages based on the notion of polymorphic invariance. This implies that the strictness property derived from one monomorphic instance of a polymorphic function applies to all possible monomorphic instances. Hughes [47] proposed a method for extending abstract interpretation to first-order polymorphic functions by calculating approximations to abstract functions of all instances from the abstract function of the simplest monomorphic instance. Abramsky and Jensen [3] showed a proof of the polymorphic invariance of strictness analysis based on the categorical notions of relators and transformations.

9.2 Summary

One of the major overheads that incur in implementing functional languages is the storage management overhead due to dynamic allocation and automatic reclamation of indefinite-extent storage. The goal of this thesis was to compute information about the lifetime of dynamically-allocated objects in higher-order, polymorphic, (either strict or non-strict) functional languages with non-flat domains, through semantics-based compile-time analyses of high-level source programs, and to use such statically inferred information for reducing the storage management overhead in functional language implementations.

In a higher-order functional language, exact information about the lifetime of objects, such as arguments and local objects defined within a function, with respect to the lifetime of the activation of the function call is generally unknown at compile-time. Thus, when storage is needed to be allocated for such object, it is usually allocated from a heap and has to be reclaimed using some kind of automatic reclamation methods. Lifetime information, if inferred at compile-time, can be useful for efficient management of storage allocated for those objects at run-time. In Chapter 2, we have presented a method for computing safe information about the relative lifetime of arguments and local objects defined within a function with respect to the lifetime of an activation of the function for a monomorphic,

strict, higher-order functional language. This method is based on a compile-time semantic analysis called *escape analysis* which provides information about the lifetimes of objects A method for improving the precision of escape information is also presented using the position information of objects in a list structure.

For structured objects such as lists and trees, the escape information that is obtainable through the escape analysis is rather coarse because it does not specify how much extent of the object escapes even when only some part of structured object escapes. In Chapter 3, we have developed a method for computing more refined escape information for a monomorphic, strict, higher-order functional language. This method is based on a compile-time semantic analysis called *refined escape analysis* which is an extension of the escape analysis and indicates how much of an object outlives the activation of the function call.

In a higher-order functional language, exact information about the lifetime of a dynamically created reference to a heap-allocated object is generally unknown at compile-time. Such information, if inferred at compile-time, can be useful for improving the reference counting scheme for both uniprocessor and multiprocessor environments. In Chapter 4, we have described a method for computing at compile-time safe information about the relative lifetime of dynamically created references to objects. This method is based on a compile-time semantic analysis called *reference escape analysis*.

In lazy evaluation, arguments to a function are not evaluated unless and until their values are demanded, and are evaluated only once upon the first demand. Their values are then saved to be used for subsequent demands, thus avoiding reevaluation. Exact information about the strictness of arguments, the order of evaluation among the arguments, and the evaluation status of arguments when demanded is generally unknown at compile-time. If inferred at compile-time, such information can be useful for a number of optimizations for lazy evaluation. In Chapter 5, we have presented a method for statically inferring a range of information including strictness, evaluation-order, and evaluation-status information in a monomorphic, higher-order, non-strict (with lazy evaluation) functional language. This method is based on a compile-time analysis called *order-of-demand analysis* which provides safe information about the order in which the values of bound variables are demanded.

All of the semantic analysis presented in the preceding chapters have dealt with a higher-order functional language with *monomorphic* type system in which every expression is rigidly typed. Most modern functional languages adopt a rich polymorphic type system which is more flexible. In Chapter 6, we have described a method, based on the notion of *polymorphic invariance*, for applying the escape analysis, the reference escape analysis, and the order-

of-demand analysis for a monomorphic language to a polymorphic language.

In Chapter 7, we have extended the escape analysis and the reference escape analysis for a strict language to a non-strict (either with normal-order evaluation or with lazy evaluation) language. Based on a source-to-source transformation of non-strict programs, we have described a method to extend the escape analysis and the reference escape analysis for a strict language to a non-strict language with *normal-order evaluation* using the analysis techniques for a strict language. The lazy evaluation model is identical to normal-order evaluation model in the standard semantics, but not in its operational semantics. We then presented the escape analysis and the reference escape analysis of non-strict functional languages with *lazy evaluation* based on the evaluation status information statically inferred by the order-of-demand analysis presented in Chapter 5.

Finally, in Chapter 8, based on the statically inferred escape information, we proposed a variety of optimization techniques to reduce the storage management overheads in functional language implementations. These include stack allocation, explicit reclamation, in-place reuse of garbage cells, reference counting elimination, block allocation/reclamation, and improving generational garbage collection.

9.3 Future Work

We would like to observe how a combination of these analyses and optimizations described in this thesis work when they are implemented in real compilers. Future work includes extensions of escape analyses to user-defined types (e.g. trees, etc.) and investigations of more efficient algorithms for finding fixpoints or of using a type system. Effective extension of the order-of-demand analysis to functional languages with lists and investigation of its applications for optimizations in lazy evaluation would be useful.

For practical purpose, it will be worth investigating a method for determining at compile-time whether the stack allocation optimization is safe in the sense that such optimization does not convert a program that runs robustly into one that does not.

It might also be fruitful to develop other semantic analyses for higher-order functional languages based on the framework which is used in the semantic analyses presented in this thesis.

Bibliography

[1] S. Abramsky. Strictness analysis and polymorphic invariance. In *Proceedings of Workshop on Programs as Data Objects,* pages 1-24, LNCS 217 Springer-Verlag, 1986.

[2] S. Abramsky and C.L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[3] S. Abramsky and T.P.Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages,* pages 49-54, 1991.

[4] L. Allison. *A Practical Introduction to Denotational Semantics.* Cambridge University Press, Cambridge, 1986.

[5] A. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters,* (25):275-279, 1987.

[6] A. Appel and D.B. MacQueen. A standard ML compiler. In *Proceedings of the 1987 Functional Programming Languages and Computer Architecture Conference,* pages 301-324, LNCS 274 Springer-Verlag 1987.

[7] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM,* 21(8):613-641, 1978.

[8] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[9] J. Barth. Shifting garbage collection overhead at compile time. *Communications of the ACM,* 20(7):513-518, July 1977.

[10] D.I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of PARLE Parallel Architectures and Languages Europe II,* pages 176-187, LNCS 259 Springer-Verlag, 1987.

[11] A. Bloss. *Path Analysis: Using Order-of-Evaluation Information to Optimize Lazy Functional Languages.* Ph.D. Thesis, Yale University, 1989.

[12] A. Bloss and P. Hudak. Variations on strictness analysis. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* pages 132-142, 1986.

[13] A. Bloss and P. Hudak. Path semantics. In *Mathematical Foundations of Programming Language Semantics,* pages 476-489, LNCS 298 Springer-Verlag, 1987.

[14] A. Bloss, P. Hudak and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation,* 1:147-164, 1988.

[15] A. Bloss, P. Hudak and J. Young. An optimizing compiler for a modern functional language. *The Computer Journal,* 31(6):152-161, 1988.

[16] R.A. Brooks, R.P. Gabriel and G.L.Steele Jr. An optimizing compiler for lexically scoped LISP. In *Proceedings of the SIGPLAN Symposium on Compiler Construction,* pages 261-275, 1982.

[17] G.L. Burn. A relationship between abstract interpretation and projection analysis. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages,* pages 151-156, 1990.

[18] G.L. Burn, C.L. Hankin and S. Abramsky. Strictness analysis of higher-order functions. *Science of Computer Programming,* 7:249-278, 1986.

[19] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys,* 17(4):471-522, 1985.

[20] D. Chase. *Garbage Collection and Other Optimizations.* Ph.D. Thesis, Rice University, 1987.

[21] D. Chase. Safety consideration for storage allocation optimizations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pages 1-9, 1988.

[22] D. Chase, M. Wegman and F. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation,* pages 296-310, 1990.

[23] C. Clark and S.L. Peyton Jones. Strictness analysis - a practical approach. In *Proceedings of the 1985 Functional Programming Languages and Computer Architecture Conference,* pages 35-49, LNCS 201 Springer-Verlag, 1985.

[24] J. Cohen. Garbage collection of linked data structures. *Computing Surveys,* 13(3):341-367, 1981.

[25] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages,* pages 238-252, January 1977.

[26] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages,* pages 238-252, 1979.

[27] L. Damas and R. Milner. Principle type schemes for functional languages. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages,* pages 157-168, 1982.

[28] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages,* pages 157-168, 1990.

[29] L.P. Deutsch and D.G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM,* 19(9), pages 522-526, 1976.

[30] M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation-order and its application. In *Proceedings of the 1990 ACM Symposium on Lisp and Functional Programming,* pages 242-250, Auguat 1990.

[31] A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[32] B. Goldberg. Generational reference counting: a reduced-communication distributed storage reclamation scheme. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* pages 313-321, 1989.

[33] B. Goldberg. Detecting sharing of partial applications in functional languages. In *Proceedings of the 1987 Functional Programming Languages and Computer Architecture Conference,* pages 408-425, LNCS 274 Springer-Verlag, 1987.

[34] B. Goldberg and Y.G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *Proceedings of the 3rd European Symposium on Programming,* pages 152-160, LNCS 432 Springer-Verlag, 1990.

[35] K. Gopinath and J. Hennessy. Copy elimination in functional languages. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages,* pages 303-314, January 1989.

[36] C.V. Hall. *Strictness Analysis Applied to Programs with Lazy List Constructors.* Ph.D. Thesis, Indiana University, 1987.

[37] C.V. Hall and D.S. Wise. Compiling strictness into streams. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages,* pages 132-143, January 1987.

[38] P. Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall, 1980.

[39] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming,* pages 351-363, Auguat 1986.

[40] P. Hudak. The conception, evolution, and application of functional programming languages. *ACM Computing Surveys,* 21(3):359-411, 1989.

[41] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages,* pages 300-314, 1985.

[42] P. Hudak and D. Kranz. A combinator-based compiler for a functional language. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages,* pages 122-132, 1984.

[43] P. Hudak and J. Young. Higher-order strictness for untyped lambda calculus. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages,* pages 97-100, January 1986.

[44] P. Hudak and J. Young. Collecting interpretations of expressions (without powerdomains). In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages,* pages 107-118, January 1988.

[45] R.J.M. Hughes. Strictness detection in non-flat domains. In *Proceedings of Workshop on Programs as Data Objects,* pages 42-62, LNCS 217 Springer-Verlag, 1986.

[46] R.J.M. Hughes. Backward analysis of functional programs. *IFIP Workshop on Partial Evaluation and Mixed Computation,* Bjorner, Ershov and Jones, editors, North-Holland, 1987.

[47] R.J.M. Hughes. Abstract interpretation of first-order polymorphic functions. In *Glascow Workshop on Functional Programming,* University of Glasgow, Department of Computing Science, August, 1988.

[48] S. Hunt. Frontiers and open sets in abstract interpretation. In *Proceedings of the 1989 Functional Programming Languages and Computer Architecture Conference,* pages 1-11, 1989.

[49] K. Inoue, H. Seki and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transaction on Programming Languages and Systems,* 10(4):555-578, 1988.

[50] T.P. Jensen and T. Morgensen. A backward analysis for compile-time garbage collection. In *Proceedings of the 3rd European Symposium on Programming,* pages 227-239, LNCS 432 Springer-Verlag, 1990.

[51] S.B. Jones and D. Le Metayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 1989 Functional Programming Languages and Computer Architecture Conference,* pages 54-74, 1989.

[52] N. Jones and S. Muchnick. Binding time optimization in programming languages: an approach to the design of an ideal language. In *Proceedings of the 3rd ACM Symposium on Principles of Programming Languages,* pages 77-94, January 1976.

[53] N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages,* pages 296-306, January 1986.

[54] U. Kastens and M. Schmidt. Lifetime analysis for procedure parameters. In *Proceedings of the 1st European Symposium on Programming,* pages 53-69, LNCS 213 Springer-Verlag, 1986.

[55] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* Ph.D. Thesis, Yale University, 1988.

[56] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams. Orbit: an optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* pages 219-233, 1986.

[57] C.W. Lermen and D. Mauer. A protocol for distributed reference counting. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming,* pages 343-350, 1986.

[58] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM,* 26(6):419-429, June 1983.

[59] C. Martin and C. Hankin. Finding fixed points in finite lattices. In *Proceedings of the 1987 Functional Programming Languages and Computer Architecture Conference,* pages 426-445, LNCS 274 Springer-Verlag, 1987.

[60] R. Milner. A theory of type polymorphism in programming. *Journal of Computers and System Sciences,* 17:348-375, 1978.

[61] J.C. Mitchell and R. Harper. The essence of ML. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages,* pages 28-46, January 1988.

[62] S. Muchnick and N. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[63] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[64] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica,* 18:265-287, 1982.

[65] Y.G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the 1st ACM Partial Evaluation and Semantics Based Program Manipulation*, pages 178-189, 1991.

[66] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[67] S.L. Peyton Jones. FLIC - a Functional language intermediate code. In *SIGPLAN Notices*, 28(8):30-48, 1988.

[68] S.L. Peyton Jones and C. Clark. Finding fixpoints in abstract interpretation. In *Abstract Interpretation of Declarative Languages,* S. Abramsky and C.L. Hankin, editors, pages 246-265, 1987.

[69] J.C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development,* pages 97-138, LNCS 185 Springer-Verlag, 1985.

[70] M. Rudalics. *Multiprocessor List Memory Management.* Ph.D. Thesis, Johannes Kepler University, Austria, 1988.

[71] C. Ruggieri. *Dynamic Memory Allocation Techniques Based on the Lifetimes of Objects.* Ph.D. Thesis, Purdue University, 1987.

[72] C. Ruggieri and T.P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages,* pages 285-293, January 1988.

[73] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transaction on Programming Languages and Systems,* 7(2):299-310, 1985.

[74] D.A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* Allyn and Bacon Inc., 1986.

[75] D.A. Schmidt. *Detecting Stack-Based Environments in Denotational Definitions.* Research Report TR-CS-86-3, Kansas State University, 1986.

[76] J.T. Schwartz. Optimization of very high level languages - I. Value transmission and its corollaries. *Journal of Computer Languages,* 1:161-194, 1975.

[77] P. Sestoft. Replacing function parameters by global variables. In *Proceedings of the 1989 Functional Programming Languages and Computer Architecture Conference,* pages 39-53, 1989.

[78] G.L. Steele Jr. *RABBIT: A Compiler for SCHEME.* Technical Report, Massachusetts Institute of Technology, 1978.

[79] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, 1977.

[80] P. Wadler. Strictness analysis on non-flat domains(by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages,* S. Abramsky and C.L. Hankin, editors, pages 266-275, Ellis Horwood, 1987.

[81] P. Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Proceedings of the 1987 Functional Programming Languages and Computer Architecture Conference,* pages 385-407, LNCS 274 Springer-Verlag, 1987.

[82] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architecture. In *Proceedings of PARLE Parallel Architectures and Languages Europe II,* pages 432-443, LNCS 259 Springer-Verlag, 1987.

[83] J. Young. *The Theory and Practice Semantic Program Analysis for Higher-Order Functional Programming Languages.* Ph.D. Thesis, Yale University, 1989.