Compiler-Directed Storage Reclamation Using Object Lifetime Analysis

James E. Hicks, Jr.

MIT/LCS/TR-555
October 1992

Compiler-Directed Storage Reclamation Using Object Lifetime Analysis

James E. Hicks, Jr.

Technical Report MIT/LCS/TR-555
October 1992

Motorola Cambridge Research Center
One Kendall Square, Building 200
Cambridge, Massachusetts 02139

Abstract

Many heap-oriented languages such as Lisp and Id depend on run-time garbage collection to reclaim storage. Garbage collection can be a significant run-time expense, especially for functional languages that tend to allocate structures often. Compiler-directed storage reclamation reduces the run-time overhead of garbage collection by having the compiler insert deallocation code. Compilers must perform object lifetime analysis in order to insert storage reclamation code. Current approaches to lifetime analysis assume a strict or sequential interpreter.

We formulate an operational semantics for a parallel, non-strict language in order to precisely define when it is safe to deallocate an object. Our operational semantics yields exact information about what objects are allocated, deallocated, and referenced at any point during the execution of a program. Using this information, we define precise run-time conditions that must be met by safe deallocation commands.

We use abstract interpretation to yield at compile-time a summary of what objects are allocated and reachable at any point in a program. We define static conditions that must be met by safe deallocation commands. We then define an algorithm that uses the abstract interpreter to verify the safety of deallocation commands already in programs and an algorithm to insert safe deallocation commands into programs.

We describe our implementation of the lifetime analysis, the verification algorithm, and the insertion algorithm. We the discuss the effectiveness of the compiler at verifying and inserting deallocation commands in several medium-sized Id programs. We also discuss the performance of each program in terms of storage allocated and reclaimed. Our implementation is quite effective for programs with simple patterns of sharing between objects.

Acknowledgments

I would like to thank Professor Arvind, my thesis advisor, for his efforts to ensure this work is first-rate. We spent many hours together, successively refining the ideas presented in this thesis. As Nikhil once said, "Fixpoint iteration is one path to nirvana." I do not think that I am there yet. I would also like to thank my thesis readers: Professors Nikhil and Weihl, for their ideas and their patience. They encouraged me to broaden the focus of my work beyond the scope of Id and Monsoon.

I would like to thank John Hughes for the discussions he had with Arvind and I that greatly clarified our understanding of abstract interpretation and its relation to lifetime analysis.

I would like to thank Olaf Lubeck for his cooperation in the early portion of my research. We began this work on storage management so that he could run large programs on Monsoon. Examination of his programs led us to formulate the safety conditions for deallocation commands. Thanks to Jonathan Young for starting me down the abstract interpretation path and for helping me understand some of its thornier issues.

I would like to thank all of the other people who read various drafts of this thesis, including Sharon Chang, Michael Ernst, R. Paul Johnson, Shail Aditya, and Arthur Lent. They have all helped to make this a coherent document.

Paul Barth has been a source of great support as he and I worked our way together through thesis writing, thesis defense, and thesis rewriting.

I could not have done this work without the support of the rest of Computation Structures Group. I have enjoyed working with all of them for the past five years. I especially want to thank all those who accepted the responsibilities that I delegated so I could finish my thesis, including Shail Aditya, Boon Ang, Alex Caro, Derek Chiou, Christine Flood, R. Paul Johnson, and Yuli Zhou. If only one of them would take over the Id Compiler. Thanks to Andy Boughton, who took me on as an undergraduate researcher years ago (doing hardware) and who has continued to put faster and faster computers on my desk ever since. My thanks also to the old guard: David Culler, Steve Heller, Bob Iannucci, Greg Papadopolous, Richard Soley, and Ken Traub for inspiration, advice and encouragement along the road to graduation. Thanks also to the other members of CSG.

Finally, I would like to thank my family. Special thanks to Sharon Chang, my wife — she has been wonderfully supportive and understanding throughout this whole endeavor. Thanks also to my parents who have always encouraged me on to bigger and better things.

— *To Sharon*

Contents

List of Figures

Chapter 1

Introduction

Many modern programming languages are oriented towards dynamic storage management. Lisp-like languages and functional languages such as Id [35, 33, 34] are heap-oriented languages. In these languages the storage for arrays and other aggregate objects is not necessarily associated with the invocation of the particular procedure that allocated the object. Storage for an aggregate is allocated on a heap when the aggregate is created so objects can last longer than the procedure that created them. The storage in which an object resides can only be reclaimed when the program no longer uses the object, where a *use* of an object is a reference to the contents of the object. The *lifetime* of an object is the period of time from the object's allocation until the time when the last reference is made to the object.

Storage in heap-oriented languages is often reclaimed implicitly, by a *garbage collector*. A garbage collector intermittently or incrementally traverses the heap, stacks, and other program data structures to determine which objects are reachable by the program, and then clears all other objects (the garbage) from the heap.

The standard alternative to garbage collection, explicit storage management, requires the programmer to insert commands to reclaim storage so that the program does not run out of memory. Explicitly deallocating structures is often an extremely error-prone process, because it is not always clear where a structure is passed and when it is no longer referenced. Furthermore, changes to a program can cause deallocation commands to become incorrect.

It is easier to develop correct programs when using implicitly managed storage, because the programmer does not have to worry about storage management. Unfortunately, implicit storage management is typically more expensive than explicit storage management. One source of overhead in implicit storage management is the determination of which storage is no longer in use. Another source of overhead is that garbage collected systems typically use more storage at any point in time than explicitly managed systems because they reserve a significant fraction of memory (up to half) for use solely during the garbage collection process. Furthermore, storage is usually not reclaimed as soon as it ceases to be used in a garbage collected system, and so more storage is allocated at any point in time than in an explicitly managed system.

In this thesis, we present a third possibility: implicitly managed storage without all of the run-time overhead of garbage collector managed storage. We accomplish this by having the compiler analyze programs and insert storage deallocation commands, thus lifting the much of the burden of storage management from the programmer. The compiler will not be able to pick up all of the garbage, and so the rest will have to be handled by the programmer or by a run-time garbage collector. In this thesis we develop and evaluate a method of static analysis of programs to determine object lifetimes. Our goal is to determine if this analysis is useful and to determine to what extent storage management can be done by the compiler. General issues of garbage collection and heap-allocation algorithms are orthogonal to this work.

This thesis makes a number of contributions to the area of lifetime analysis and to the practice of static program analysis. First, it develops a general framework for the lifetime analysis of a parallel language using the theory of abstract interpretation. Second, it defines abstract representations for a variety of data types, including tuples, arrays, algebraic types, recursive types, and higher-order functions. Third, it attempts to characterize the costs and effectiveness of these techniques when applied to real programs.

Past work in lifetime analysis has mostly been on sequential languages. We know of no work which performs lifetime analysis on either sequential or parallel non-strict languages. The lifetime analysis method described in this thesis applies to parallel, strongly-typed, single-assignment languages. Slight variations in the methods we use allow us to analyze either strict or non-strict programs. We present the work in terms of a non-strict language and discuss the changes necessary to apply our methods to a strict language.

Our main goal was to develop a framework for lifetime analysis and to determine its effectiveness. In fact, we have developed a general framework for abstract interpretation of parallel and non-strict languages with a rich variety of types. This abstract interpreter could be used to perform interference analysis or even strictness analysis instead of lifetime analysis, although we have not pursued these topics. We do consider a limited form of sharing analysis to determine if the elements of arrays may be shared. We also discuss extensions to the lifetime analysis of recursive types that would allow us to determine whether objects form directed acyclic graphs or trees.

We have found our implementation of these methods to be quite effective in determining the lifetimes of objects in real Id programs. We have implemented this work as part of the Id compiler [40] and applied it to several programs of 100 to 1000 lines. The implementation is structured to support separate compilation — a program can be compiled in bottom up fashion and each procedure is verified/transformed individually. The augmented compiler was able to compile these programs into object code that deallocated 80 to 100 percent of the total storage that they allocated, at a cost of a factor of 1.5 to 5 increase in compile-times. We have also found that although a programmer could insert all of the deallocation code that the compiler inserted, it would require a major change in programming style to do so.

1.1 Thesis Overview

Before we can develop a lifetime analysis algorithm, we must have a well-defined notion of the lifetime of an object. We defined the lifetime of an object to be the period of time during the execution of a program from when the object was allocated until the object was no longer referenced. Lifetime analysis is the process of determining the range of program points during which the object bound to a particular variable may be referenced by a program. Thus, lifetime analysis is intimately related to the operational semantics of a program.

In this thesis, we first develop an operational semantics for a non-strict, parallel language. The operational semantics is defined by an interpreter that gives the standard semantics of a program in terms of its behavior. We then use this semantics to define the lifetimes of objects allocated by programs. Our definition of object lifetimes is exact, but can only be determined at run-time, as a program is executing. We also use this interpreter to define when deallocation commands are correct. Correct deallocation commands never lead to run-time errors in which an object is deallocated before the end of its lifetime.

Because we must be able to determine object lifetimes at compile-time, we need some way of converting this exact, run-time notion of object lifetimes into a compile-time property. We use an abstraction of the standard interpreter to give an approximation of the behavior of the program. The purpose of the abstract interpreter is to generate an approximation of a program's behavior over all input data, and, in the case of parallel programs, over all execution orders. In addition, the abstract interpreter must be decidable — we must be able to compute this approximate behavior in a finite amount of time.

Given the abstract interpretation of a program, we show how to compute approximate object lifetimes. We are willing to use approximate object lifetimes in order to develop an algorithm that terminates, as long as the approximations are all safe. A safe approximation of an object's lifetime is guaranteed to include the actual lifetime of an instance of that object at run-time.

We present two algorithms that use the information about object lifetimes. The first algorithm verifies that the deallocation commands in a program are all safe. The second algorithm inserts safe deallocation commands into programs automatically. This second algorithm allows the programmer to write programs in which storage is implicitly reclaimed.

For the remainder of this thesis, we talk about the language KID^- [3], a specific parallel, non-strict, single-assignment language with higher-order functions. KID, or Kernel Id, is an intermediate language developed by Ariola and Arvind to express the semantics of Id [35, 33, 34] and to express the compilation of Id programs. In this thesis, we consider KID^- to be KID without higher-order functions and M-structures [7] (structures with per-element mutual exclusion).

In Chapter 2 we present the syntax and standard semantics of KID^- and we discuss the unusual evaluation strategy used by the KID^- interpreter. In Chapter 3 we develop an augmented, or *instrumented*, interpreter that allows us to define exactly when deallocation commands are correct and incorrect.

In Chapters 4 and 5 we restrict KID^- programs to operate only on tuples, numbers, and booleans. In the first of these chapters we develop an abstracted interpreter for KID^- and show that it is *safe* with respect to the instrumented interpreter. Our definition of safety is that object reachability must be preserved by the abstract interpreter. In the second of these chapters we use the abstracted interpreter to give an algorithm for verifying safe deallocation commands and an algorithm for inserting deallocation commands. In Chapter 6 we discuss a method for improving the effectiveness of the lifetime analysis by improving the abstract interpreter.

In Chapters 7, 8, and 9 we describe the additions to the abstract interpreter necessary to handle arrays, algebraic types, recursive types, and higher-order functions.

In Chapter 10, we describe our implementation of the deallocation command verification and insertion algorithms and their effectiveness on several programs. Finally, we give our conclusions on this work in Chapter 11.

The remainder of this chapter gives some background on lifetime analysis and storage management. Section 1.2 describes previous work relating to lifetime analysis. Section 1.3 describes the assumptions we make about storage management. Section 1.4 compares the cost of garbage collection with the cost of explicit storage management. Section 1.5 describes explicit storage management in Id programs. Finally, Section 1.6 describes the safety condition that must be met by deallocation commands in Id programs.

1.2 Background

This section gives some background on the problem of storage management. We start by describing the various storage management strategies, and then we go into more detail about the techniques used in implicit storage management. There are several ways in which each technique could be classified, and so the division of techniques into groups is somewhat arbitrary.

The problem of storage management has existed since the first computer program was written. In early programming languages such as Fortran, storage is statically allocated by the programmer. Under the static management paradigm, the programmer or compiler allocates storage for all structures by creating a memory map that places each object in a fixed position. In early Fortran implementations, all procedure activations and all data structures were statically allocated. In modern computer languages, some data structures may be statically allocated. There is no direct run-time cost for the management of statically allocated storage — this all occurs at compile-time when the memory map is constructed.

Static allocation is not always possible: the activation frames for recursive procedures cannot be statically allocated. A separate activation frame must be allocated for each recursive procedure invocation. For this reason, procedure activation frames, including storage for procedure-local objects, are usually stack managed. Temporary structures, declared locally in procedures, may also be stack allocated.

Under stack management, storage is managed by having a pointer to the next word to be allocated, incrementing this pointer to allocate storage and decrementing this pointer to deallocate storage. Under stack discipline, objects must be deallocated in the reverse order from which they were allocated (last-in-first-out). Stack management allows greater flexibility than static allocation, because the number and size of objects does not have to be known at compile-time.

Objects allocated on the same stack as activation frames are automatically deallocated when the procedure that allocated them returns to its parent. If a pointer to this object is returned to the parent procedure, the parent may attempt to refer to the contents of a defunct structure. This scenario is known as the dangling pointer problem. The danger is that the object may be overwritten when another procedure call is made, and that the parent will thereafter read spurious data.

Sometimes it is necessary for objects to survive longer than the procedures that allocated them. In this case, they must be handled by a heap management algorithm that allows objects to be allocated and deallocated in an arbitrary order. Heap management increases the expressiveness of a language but complicates the storage manager by making it more expensive computationally. The heap manager must keep track of which storage is in use and which storage is free to be allocated, while trying to minimize wasted storage due to mismatches between the sizes of objects requested and the sizes of objects actually allocated.

In many languages that support heap allocation, such as C, both allocation and deallocation must be specified by the programmer. If the programmer does not deallocate structures that are no longer needed, then the program consumes an inordinate amount of memory, possibly causing the program to fail. If the programmer deallocates an object too soon, then the program may behave incorrectly due to a dangling pointer error.

The explicit deallocation of structures whose lifetime is not tied to that of a procedure invocation is difficult — the programmer must ensure that no more references to an object are made anywhere in the program. The difficulty is increased if the pattern of sharing among objects is complex.

For this reason, many heap-oriented languages have run-time system support to automatically (or implicitly) deallocate storage that is no longer in use.

In heap-oriented languages such as Lisp, storage management is implicit: all structures allocated are typically allocated on a heap, and the run-time system automatically takes care of deallocating structures that are no longer accessible from the program. Structures that are not accessible from the program are called garbage. The garbage collector, a part of the run-time system, periodically scans the heap and invocation stacks, and finds and reclaims all unreachable objects. In general, garbage collection is more expensive than explicit heap management; in addition to reclaiming storage the garbage collector must determine which objects are garbage. The benefit from using a garbage collected system is that the user does not have to worry about not deallocating enough storage or about deallocating storage too early.

1.2.1 Lifetime Analysis

Lifetime analysis was first suggested by Barth [5] as an optimization to shift some of the run-time overhead of garbage collection to compile-time. His approach is to take Lisp programs that had reference counting code inserted, and to use dataflow (live variable) analysis to determine that a particular variable in the program will always be associated at run-time with a structure with reference count 1. When a variable is determined to be dead, then code can be inserted to free the associated structure. Barth also discusses several local transformations that optimize reference counting code inserted by the compiler. Although his method only inserts deallocation code if it determines that there is exactly one reference to a structure, he claims that this optimization is powerful enough to reclaim a significant amount of temporary storage in Lisp programs, because studies by Clark [11] show that most structures in Lisp programs are referred to exactly once.

1.2.2 Dataflow Analysis

Barth's method was limited because the analysis could not follow pointers or procedure calls. There have been several approaches that attempt to solve these problems.

Ruggieri and Murtagh [39] developed an interprocedural lifetime analysis framework for a statically typed, monomorphic language. Their algorithm computes the set of object sources which may be bound to each variable before each statement in the program is executed. They represent nested objects as subvariables, with labeled edges connecting variables with the contents of their various fields. Recursively typed objects have a potentially infinite number of subvariables; so Ruggieri and Murtagh introduce an operator that summarizes an infinite graph of subvariables by one in which the longest path is bounded by n, where n is a parameter of the analysis.

Larus and Hilfinger [29] developed an analysis similar to Ruggieri's which computes the possible aliases between structure accesses. They show how to use standard dataflow techniques to compute their *alias graphs*. They also show that precise computation of alias relations in a single function is NP-complete.

Hendren and Nicolau [20] take a different approach to solving the finite representation problem. They define an analysis framework that uses *path matrices* to do interference analysis for parallelization. These path matrices show the paths of possible interference between two successive program points. Each element of a path matrix uses a regular expression of field names to name an access path through a recursively typed object. This naming scheme guarantees that access paths are of finite size. Hendren and Nicolau's method automatically detects non-shared lists and trees in an imperative language. The interference analysis Hendren and Nicolau developed can be recast as a lifetime analysis by determining all the statements from which a given structure is reachable — the control region bounded by those statements bounds the lifetime of the structure.

Chase, Wegman and Zadeck [10] attempt to improve the method by which information about data structures is summarized. Their method takes programs in *static single assignment* form [14] and constructs a *storage shape graph* (SSG) that represents the interconnectedness of structures in the heap. Each node in the graph represents a structure allocated by a different allocation statement. The number of nodes in an SSG is bounded by the sum of the number of allocation statements and the number of variables in a program. Storage shape graphs are augmented with heap reference counting to determine the lifetime of a structure and to determine if a structure is acyclic.

1.2.3 Analyses Based on Abstract Interpretation

These techniques all consist of a set of *ad hoc* rules for analyzing programs. Cousot and Cousot [12] developed *abstract interpretation*, a method for simulating the execution of a program in order to determine the behavior of a program. The use of abstract interpretation allows the derivation of an analysis framework from the operational semantics of a programming language.

Jones and Muchnik [27] used abstract interpretation to develop a general framework for interprocedural dataflow analysis of programs with recursive data structures. They extend the Cousots' work on dataflow analysis of flowcharts to work with recursive data structures. They use *tokens* to provide local representations of lists. Tokens are labels derived from program states. Their flow analyzer constructs a *retrieval function* that takes a token and reconstructs the list or lists locally described by that token. This retrieval function is really an abstraction of a store, where a store maps locations to list values.

Jones and Muchnik describe a version that analyzes a simple first-order language. This version uses node labels as tokens, and divides tokens into atoms and lists. Their general framework could be adapted to a variety of analyses by plugging in the appropriate domains and operational semantics. There is a great deal of freedom in choosing tokens. Tokens can be more specific, *e.g.*, whole states, in which case the analysis will be more precise but computationally intractable, or more general, *e.g.*, node labels, in which case the analysis will converge faster but give less precise information.

Horwitz, Pfeiffer, and Reps [22] use the Jones and Muchnik framework to compute an abstraction of memory where each location is labeled by the program points that modify its contents. They show that their analysis is correct for all implementations of the underlying operational semantics. The framework of Horwitz *et al* does not do interprocedural analysis.

One enhancement to this framework is the ability to handle higher-order functions. Deutsch [15] develops a static analysis method for determining the aliasing and lifetimes of objects in a strict, higher-order functional language with first class continuations. His work is also based on that of Jones and Muchnik. Deutsch presents a low-level operational semantics defined in terms of state transition rules, and abstracts this semantics to obtain an analysis algorithm. He uses complete program states to label objects uniquely in the standard semantics and uses an abstraction of program states to label objects in the abstract semantics.

Rather than presenting a low-level operational semantics, Harrison [19] presents an analysis in terms of a high-level operational semantics for Scheme. Harrison develops an analysis that could be used to make storage management and parallelization decisions about Scheme programs with first class continuations, side effects and higher-order functions. His work takes an approach similar to that of Jones and Muchnik. The correct modeling of control flow in the presence of continuations adds to the complexity of Harrison's method. He uses *procedure strings* to name all points in the execution of a program. A procedure string consists of a sequence of symbols naming the procedure bodies that have been entered and exited along the execution path to a program point. Harrison models aggregate objects as higher-order functions.

1.2.4 Analysis of Parallel Languages

All of these techniques were developed for sequential programming languages, even though the original work on abstract interpretation was defined in terms of flow graphs, which are not necessarily sequential. Much of the work on abstract interpretation has been done on functional languages, which are often touted as being parallel languages. Even so, most of the work on lifetime analysis of functional languages has been done with respect to a sequential implementation. There have been a few approaches that do not assume a sequential implementation, which we will describe below.

Hudak [23] describes an analysis based on abstract interpretation of a reference counting interpreter for a strict, functional language operating on arrays of numbers. Even though the language is functional, the denotational semantics he presents is sequential, because it performs side effects in the form of reference counting operations.

Thomas Johnsson [26] developed an analysis method for modeling heap contents based on the framework of Jones and Muchnik. His analysis is to be used in optimizing graph reduction intermediate code that resulted from compiling a lazy, functional language. Although the language being compiled is not sequential, the interpreter of the intermediate code is sequential. The intermediate code is imperative and contains explicit code to construct and evaluate closures. The parallelism in the source language is simulated by interleaving execution of subexpressions in the intermediate code.

Ranelletti [38] describes an analysis method on dataflow graphs representing parallel programs written in SISAL [16]. These dataflow graphs only give a partial order on the execution order of expressions in a program. This method allows the compiler to transform graphs so that storage is preallocated for arrays that are incrementally defined by a program. Preallocation reduces the number of arrays that need to be allocated and reduces the number of times array elements are copied from one array to another. Ranelletti's method is very efficient — it takes $O(n)$ compile-time, where n is the size of the program being analyzed. Unfortunately, extending it to handle interprocedural analysis will make it much less efficient — it will take $O(2^n)$ compile-time.

Cann [9] describes an analysis technique on SISAL dataflow graphs that allows arrays or array dope-vectors to be updated in place whenever it can be shown that the updater is the only consumer of the array. This method is also based on parallel programs. However, some of his transformation techniques add dependence edges that increase the sequentiality of the program in order to perform update-in-place optimizations.

In addition to these graph-based approaches, there have been a number of abstract interpretation-based analysis frameworks that are interesting because they also do not assume a sequential interpreter. The work by Young and O'Keefe [45] and the work by Aiken and Murphy [1] fall into this category.

Young and O'Keefe developed a type evaluator for a lazy dialect of Scheme. This evaluator computes an approximation to the set of possible values to which each expression in a program could evaluate.

The only data structure that they considered was untyped pairs. In order to analyze recursive functions on lists, their analyzer approximated infinite sets of values by cyclic type representations. Although the evaluator described by Young and O'Keefe yields an approximation of the values computed by each expression in a program, it is not viable for use in lifetime analysis because there is no way to determine the sharing or reachability of objects from any expression in the program.

Aiken and Murphy developed a similar type inferencer for the strict functional language FL. Their approach uses type expressions as the abstract value domain and a set of rewrite rules to give the operational semantics of FL. The language of type expressions includes a *fix* operator that defines an infinite set of regular tree types by a finite representation. These recursive type expressions are used when deriving the type of recursive functions.

Aiken and Murphy's type inferencer uses the rewrite rules as constraints in a proof system to derive the types of FL expressions. In the case of recursive functions, heuristics must be used to choose which rewrite rule to apply, because more than one rewrite rule may be applicable to a given instance of a recursive function.

Park and Goldberg [37] developed an analysis framework based on abstract interpretation of a higher-order functional language. Their framework computes an approximation of how much of a nested list value passed to a function escapes as part of the result of that function. They did not precisely define the standard semantics that they were abstracting.

Jones and Le Métayer [28] developed three analyses framed as abstract interpretations of programs: sharing, transmission, and necessity analysis. These analyses are defined for an expression-oriented language with lists as the only data structure. Jones and Le Métayer did not state precisely the standard semantics corresponding to the abstract semantics used in the analyses, and so it is difficult to see how to generalize this method to other data structures.

There are two problems with the last two approaches to determining object lifetimes. The first is that they do not have a good correspondence with any standard semantics. The point of methods based on abstract interpretation is that the analyses can be shown to be safe with respect to the standard semantics. The second problem is that objects are not named, and so the analyses fail if the source languages are made imperative or non-strict, because there is no way to handle cyclic structures in these frameworks.

1.2.5 Analyses Based on Type Deduction

There is one more semantics-based approach to analysis that defines the analysis in terms of type deduction or type checking using a non-standard type system. Lucassen and Gifford [31] define a type and effects system for the FX language [17] that can be used to determine the lifetimes of objects. FX-87, based on the second-order lambda-calculus, has a kind system consisting of type and effect annotations. The effect annotations describe which *regions* are allocated into, written to, or read from during the execution of an expression. Effect annotations on procedure values describe not only the effects incurred by evaluating the procedure value, but also the *latent* effects incurred by applying the procedure value to arguments. Lucassen and Gifford show how the effect descriptions can show that the lifetime of an object resulting from a particular expression has limited extent.

This approach requires the user to annotate programs with type and effect declarations before the compiler can perform type and effect checking and lifetime analysis. Use of this approach would also allow the compiler to check the safety of explicit storage management in some cases. In later work Gifford *et al.* [18] extend the FX compiler to perform type and effect deduction, but in this effect system they dropped the information about storage regions. It is unclear from the paper whether there is an efficient or decidable algorithm for deducing types and effects with regions for FX programs.

Baker [4] describes states that the structure-sharing unification algorithm for Milner-style type inference already produce a certain amount of sharing information for functional languages. Each node that represents the type of an expression in a program corresponds to a set of run-time objects. In a functional language, distinct type nodes represent disjoint sets of run-time objects, while unified type nodes represent overlapping sets of run-time objects. The advantage of using type inference for sharing analysis is that the algorithms for type inference are efficient enough to be used in production compilers. The disadvantage of this approach is that it cannot be extended to imperative programming languages without greatly increasing the complexity of the analysis.

1.3 Storage Management Assumptions

Let us assume that objects allocated by a program can be placed either in the activation frame of the procedure that allocated the object or on an implicitly or explicitly managed heap. Objects placed in procedure activation frames are automatically deallocated when the procedure terminates; consequently, the lifetime of these objects must be bounded by the lifetime of the procedure. In our implementation of Id, only fixed size objects may be frame-allocated because a procedure's activation frame cannot be extended once it has been allocated.

We believe that the applications in which we are interested would suffer too much of a performance penalty if they depended solely on run-time garbage collection. One characteristic of these applications is the use of large amounts of data, often held in large arrays. The behavior of garbage collectors in the presence of large, shallow or flat data structures is not well understood, but applications typically manage these structures explicitly even though garbage collection is used to manage other structures. In these programs, most storage reclamation should be done explicitly, either by explicit deallocation or explicit reuse of structures. We would like to automate the process of explicitly managing these large structures. It is very difficult, and often impossible, for either a programmer or a compiler to explicitly reclaim all structures allocated, and so we will continue to have a garbage collector that reclaims the storage that cannot be reclaimed explicitly.

This thesis does not explore the best ways for the heap manager and garbage collector to interact. The way explicit and implicit storage management interact depends to a large extent on the choice of garbage collection method and characteristics of the run-time system. One possibility is for the heap manager to allocate areas that are never garbage collected, and to use these areas for objects that are guaranteed to be deallocated eventually. The objects in these areas would never have to be copied by the garbage collector, and so we would save on the overhead of copying these objects. Another possibility is to use a reference counted garbage collector and to set the reference counts of objects whose lifetimes can be determined to one upon allocation and to zero when no longer needed, but not to perform reference counting operations on the objects otherwise.

1.4 Cost of Storage Management

It is not clear that a program will always have better performance running under an explicit storage manager than it will have running under a garbage collector. Appel [2] makes an argument that garbage collection can be faster than explicit storage management; he claims that it can even be faster than stack allocation. Appel's claim is that

> with enough memory on the computer, it is more expensive to explicitly free a cell than it is to leave it for the garbage collector — even if the cost of freeing a cell is only a single machine instruction.

Appel gives the cost per reachable object of copying garbage collection as

$$g = \frac{(c_1 + c_2 s)A}{M/s - A} \tag{1.1}$$

where c_1 is the number of operations required per object copied, c_2 is the number of operations per pointer, s is the average size of an object, A is the number of reachable objects when garbage collection is performed, and M is the size of the two memory spaces. If M is made sufficiently large relative to the other parameters, then the cost per reachable, or non-garbage, object can be made arbitrarily small.

In the limiting case as the amount available memory approaches infinity, Appel asserts that it is cheaper to rely on garbage collection than explicit storage management, even stack management, because the garbage collector will never have to run. At the other extreme, as the amount of memory approaches the average amount of memory in use at any time, the cost of garbage collection goes to infinity. In order to determine the crossover point where the cost of implicit memory management is less than the cost of explicit memory management, we must know the average amount of memory used by a program and the time constants c_1 and c_2 associate with garbage collection, relative to the cost of explicit storage management.

Is it reasonable to assume, as Appel does, that we will be operating in the large-memory regime where the cost of garbage collection is insignificant? Although the cost per word of memory is continuously decreasing, the amount of memory needed for interesting problems seems to be increasing just as fast. It seems that the cost of garbage collection will be significant for the class of programs considered in this thesis because large programs will operate in the memory management regime where most of memory is in use and garbage collection is expensive. Nevertheless, the cost of explicitly allocating and deallocating an object by a general heap manager is very high, and so care must be taken to reduce the number of calls to the general heap manager. For this reason, we will consider some approaches to reusing storage directly or allocating objects in procedure activation frames.

Appel does not consider the effect of locality on program execution time. Moon [32] states that the most important responsibility of a garbage collector in a system using virtual-memory is to keep data structures local; actually reclaiming storage is a secondary responsibility in this case. If a program has little locality of reference because it uses objects spread over a very large amount of memory, then the performance of the program will be very poor if the virtual-memory system thrashes.

Is there some way for explicit storage management to cooperate with garbage collection? Many of the strict, functional languages use a reference counting garbage collector because these languages cannot create cyclic data structures. If a reference counting garbage collector is used, then reference counting of objects whose lifetime is known need not be performed. The reference count will be set to one when the object is created and set to zero when the object's lifetime is over. The Id run time system is likely to use a copying garbage collector, so that it can reclaim circular objects.

```
def f0 () =
  { x = MakeTuple(6,847);
    r = Select1(x);
   ---
    Dealloc(x)
   in r }
```

Figure 1.1: Dealloc in a block expression

A explicit storage allocation and deallocation can cooperate with a copying garbage collector by allocating and deallocating objects in a separate region. Garbage collection will not be performed on this region until all objects within it are garbage, in which case, the region may be used as free storage. Alternatively, this region can be treated as an older generation, and garbage collected infrequently, with promotion suppressed.

1.5 Explicit Storage Deallocation in Id

The first step in this work was to allow programmers to perform explicit storage management in Id. We introduced an experimental feature into the language for explicit deallocation of structures. The `Dealloc` primitive, along with `---` (local barrier synchronization) allows Id programmers to insert commands that deallocate the storage associated with an object when that object is no longer in use.

Programmer-directed deallocation will be performed to determine the costs and benefits of explicit deallocation in terms of program performance and the problem sizes that may be run without exhausting memory or invoking the garbage collector.

The `Dealloc` primitive explicitly deallocates the storage associated with a structure in Id. In order to use the `Dealloc` primitive, we must have proper synchronization that prevents the `Dealloc` from executing until all uses of the structure to be deallocated have executed. For that reason, we have also introduced a barrier synchronization construct, denoted by three or more dashes: `---`.

In Id, unlike other parallel languages, a barrier is a local synchronization. A barrier can only appear within a `letrec` block, and its effects are limited to that `letrec` block. A barrier in Id ensures that the code in the block bindings before the barrier executes to termination before the code in the block bindings after the barrier. We will define a *control region* to be the program region containing a group of block bindings delimited by barriers. In Id, a control region *terminates* when all computation threads have exited the control region. In other words, all values in the region have been produced and all side effects have been performed.

The example in Figure 1.1 contains a block with one control region consisting of the bindings of `x` and `r`. In this example, the object to which `x` is bound will be deallocated when the computation in both bindings in the control region have terminated.

Invocation and termination of control regions are partially ordered. Invocation is the point in time when the interpreter first begins executing a portion of a control region, and termination is the point in time at which the interpreter finishes executing all code in a control region. Naturally, termination of any control region always occurs after invocation of that control region.

```
def f0() =
  { x = MakeTuple(6, 847);
    r = { y = Select1(x);
          z = Select2(x);
          r1 = y + z;         cr1
        in r1 }               cr0
    ---
    Dealloc(x)  cr2
  in r }
```

Figure 1.2: Statically nested control regions

Control regions may be composed by enclosing one control region within another or by placing a barrier between two control regions. If control region cr_0 statically encloses control region cr_1, then the invocation of cr_0 must precede the invocation of region cr_1 and the termination of cr_0 must follow the termination of cr_1.

Definition 1.1 (Barrier Relation) *The relation* (cr_0 `---` cr_1) *holds if control region* cr_0 *is statically separated from* cr_1 *by a barrier and* cr_0 *comes before the barrier textually.*

If control region cr_0 is separated statically from control region cr_1 by a barrier, and cr_0 comes before cr_1, then both the invocation and termination of cr_0 must precede the invocation of cr_1.

Consider the body of procedure f_0 in Figure 1.2. In this example there are three control regions: region cr_0 which is composed of the bindings of `x` and `r`, region cr_1 which is composed of the bindings of `y` and `z`, and region cr_2 which is composed of the deallocation command. Region cr_0 encloses region cr_1; therefore, the invocation of cr_0 precedes that of cr_1, and termination of cr_1 precedes that of cr_0. Region cr_0 is separated from region cr_2 by a barrier, and so both the invocation and termination of cr_0 must also precede the invocation of cr_2. The control region composition relations are transitive; therefore, the invocation and termination of region cr_1, enclosed by cr_0, must precede the invocation of region cr_2.

The ordering of the invocation and termination of dynamically composed control regions follows from that of statically composed control regions. If control region cr_0 contains a procedure call, and cr_1 is the control region of the run-time instance of the body of that procedure call, then we say that cr_0 dynamically encloses cr_1. Therefore, the invocation of cr_0 will precede the invocation of cr_1 and the termination of cr_0 will follow the termination of cr_1.

The example in Figure 1.3 is similar to Figure 1.2, except that control region cr_1 is in the body of procedure `g`. In this example, control region cr_1 is dynamically enclosed within control region cr_0 because procedure `g` is called from within control region cr_0. Therefore, the partial ordering of invocation and termination of control regions will be the same as in the previous example. Clearly, we must be able to name dynamic instances of control regions if we are going to be able to talk about the ordering of invocation and termination of those regions. This naming of dynamic instances is one of the topics we will discuss in more detail later in this thesis.

```
def f0() =
     { x = MakeTuple(6, 847);
       r = g(x);                cr0
       ---
       Dealloc(x)   cr2
       in r }
def g(x) =
     { y = Select1(x);
       z = Select2(x);
       r1 = y + z;         cr1
       in r1 }
```

Figure 1.3: Dynamically nested control regions

If one control region statically or dynamically encloses another, then the lifetime of the outer region will completely include the lifetime of the inner region. On the other hand, if two control regions are separated by a barrier, then the lifetime of the first will completely precede the lifetime of the second control region.

From these two properties we determine that control regions form a natural tree.

Definition 1.2 (Ancestor Relation) *Control region* cr_0 *is an* ancestor *of region* cr_1*, or*

$$cr_0 = cr_1 \uparrow$$

if cr_0 *statically or dynamically encloses region* cr_1*.*

In addition, we say that control region cr_1 is a *descendent* of region cr_0 if cr_0 is an ancestor of region cr_1. Every control region cr_0 will be considered to be an ancestor and a descendent of itself.

$$cr_0 = cr_0 \uparrow^0$$

The expression $cr_0 \uparrow^n$, where $n \geq 0$, refers to the nth ancestor of region cr_0.

We will now define two precedence relations on control regions.

Definition 1.3 (Invocation Precedence) *The* invocation precedence *relation* ($cr_0 \preceq_I cr_1$) *is defined as follows:*

$$cr_0 \preceq_I cr_1 \equiv \left(\begin{array}{c} \exists n.\ cr_0 = cr_1 \uparrow^n \\ \vee \\ \exists n_0, n_1.\ cr_0 \uparrow^{n_0} \text{---} \ cr_1 \uparrow^{n_1} \end{array} \right)$$

If ($cr_0 \preceq_I cr_1$), then control region cr_0 must be invoked before cr_1 may be invoked.

Definition 1.4 (Termination Precedence) *The* termination precedence *relation* ($cr_0 \preceq_T cr_1$) *is defined as follows:*

$$cr_0 \preceq_T cr_1 \equiv \left(\begin{array}{c} \exists n.\ cr_0 \uparrow^n = cr_1 \\ \vee \\ \exists n_0, n_1.\ cr_1 \uparrow^{n_0} \text{---} \ cr_0 \uparrow^{n_1} \end{array} \right)$$

If ($cr_0 \preceq_T cr_1$), then control region cr_0 must terminate before cr_1 may terminate.

1.6 Safety of Explicit Deallocation

Note that the use of the `Dealloc` primitive is inherently unsafe. The programmer may try to deallocate structures that are shared between various parts of the program, causing all sorts of errors to occur. Just as in other languages with explicit allocation and deallocation, `Dealloc` introduces the possibility of dereferencing dangling pointers. Therefore, the programmer must analyze his program to verify the safety of each explicit deallocation performed. In this section, we will see a set of informal conditions that must be met in order to safely deallocate storage in a program.

Conceptually, there is a single condition that must be satisfied in order to safely deallocate or reuse an object. An object may be deallocated when there are no further references to the object. In an implicitly managed system, an object will be deallocated when there are no live references to the object. A *live reference* to a data structure is a reference, or pointer, that is stored in either the activation frame of a procedure invocation, or in a static variable or in a data structure to which there is a live reference. In a system in which the programmer must explicitly manage storage, an object may be deallocated as soon it can be guaranteed that no further use will be made of the object, and so the lifetime of objects may be shorter than in a system with garbage collection.

One way to guarantee that there will be no further references to an object is to put the deallocation command in a control region or control region that executes after all uses of the object execute. Here is a condition that describes when it is safe to deallocate an object.

Condition 1.5 (Deallocation Safety) *Given two control regions cr_0 and cr_1 and a variable x and structure ol bound to x in both regions, it is* safe *to deallocate the structure ol in control region cr_1 if*

1. $\forall cr_r.\ \mathit{UsedIn}(ol, cr_r) \Rightarrow \exists n.\ cr_0 = cr_r \uparrow^n$
2. $cr_0\texttt{---}\ cr_1$,
3. *the only use of ol in region cr_1 is in the deallocation of ol, and*
4. *the only deallocation of ol is in region cr_1.*

where $\mathit{UsedIn}(ol, cr)$ is true if object ol is allocated or dereferenced in control region cr.

The first two subconditions of Condition 1.5 guarantees that all uses of the structure bound to variable x in control region cr_0 have terminated before the `Dealloc` in control region cr_1 can execute. The third subcondition guarantees that there are no references to the contents of x in cr_1 that may execute after x is deallocated, and the fourth subcondition guarantees that the structure bound to x is deallocated only once.

Condition 1.5 is sufficient to ensure the safety of the deallocation of structure x. This condition is very conservative; it means that the control region containing the producer and all the control regions containing consumers of the structure have terminated before the deallocation statement executes.

Figures 1.4 and 1.5 show procedures from the Gamteb [8] photon transport simulation benchmark. The procedure `compton`, shown in Figure 1.4, contains a `Dealloc` command that satisfies Condition 1.5. The variable `new_particle` is bound to a newly allocated tuple in the first binding in the body of procedure `compton`; the structure to which `new_particle` is bound is deallocated in the control region after the barrier. Note that procedure `transport_particle` uses `new_particle` but does not store it anywhere or return it as a value. Procedure `compton` allocates nine words of storage for the new particle. Adding the `Dealloc` statement allows that storage to be reclaimed as soon as `compton` terminates.

The procedure `handle_collision`, shown in Figure 1.5, contains a binding of variable `t_particle` to a structure allocated in the body of procedure `photo_elect` and used in the body of `compton`. The control regions in which this structure is allocated and used are all descendents of the control region in which `t_particle` is bound. The deallocation of the structure bound to `t_particle` in the control region after the barrier is safe because the allocation and all other uses of this structure occurred either in the control region before the barrier or in descendents of that control regions, and so all of these uses must have terminated before the deallocation command is invoked.

In the rest of this thesis, we show how to verify the safety of a deallocation command at run time. We show how to check the safety of deallocation commands at compile-time using a conservative approximation of when objects may be allocated, dereferenced, and deallocated. We also present an algorithm for inserting safe deallocation commands at compile-time.

```
def transport_particle xsect_table bins particle prob =
  { x,y,z, u,v,w, wt, e, e_bin, cell, seed = particle;
    pcompton, ppair, pphoto, ptotal = prob;
      ...
    d_surf, surface = dist_to_surface x y z u v w;
    rand, rand1 = grand seed;
    d_coll = dist_to_collision ptotal rand;
    bin_counts = if (d_coll >= d_surf) then
                   move_to_surface d_surf
                  else  % (d_coll < d_surf)
                   handle_collision d_coll;
   in
     bin_counts};

defsubst compton particle d_coll xsect_table bins =
  { %% Allocate a new particle, deallocate it in this context.
    new_particle = (new_x,new_y,new_z, ··· new_seed );
     ...
    r =
     if e_kill then
       ...
     else
       (transport_particle xsect_table bins
                           new_particle new_prob)
   ---
    Dealloc new_particle ;
   in r };
```

Figure 1.4: Procedure `compton`

```
defsubst handle_collision d_coll =
  { %% t_particle is allocated within photo_elect,
    %% and deallocated in handle_collision.
    t_particle, absorb, wt_kill =
       photo_elect particle d_coll pphoto ptotal ;
    counts =
      if (not wt_kill) and (rand1 < p_compton) then
        compton t_particle d_coll xsect_table bins
      else
    ...
    r = add_counts counts col_counts ;
   ---
    Dealloc t_particle ;
   in r }
```

Figure 1.5: Procedure `handle_collision`

Chapter 2

Problem Statement

In order for the compiler to verify or insert explicit storage deallocation code in programs, it must be able to determine the lifetimes of the objects being deallocated. Thus, the compiler must have some notion of the run-time behavior of the program being compiled. In this thesis, the compiler will use an abstraction of the operational semantics to determine the lifetimes of objects.

This thesis develops a method for lifetime analysis that is directly applicable to parallel, single-assignment languages. In particular, we will be using the language KID$^-$ as the basis for the analysis.

The first step in developing our lifetime analysis algorithm is to define a standard operational semantics for the language of interest. One can define the operational semantics in terms of an abstract machine, in terms of a term rewrite system, or in terms of an interpreter. We define the operational semantics in terms of an interpreter because that allows us to stay close to the original source code of the program, rather than compiling into object code for the abstract machine.

This chapter describes KID$^-$ syntax and gives its semantics in terms of an interpreter. In the first section, we define the notation used throughout this thesis. In the second section, we define the syntax of KID$^-$ programs and the value domains over which KID$^-$ programs operate. In the third section, we define the standard KID$^-$ interpreter and give examples of its operations. In the fourth section, we define the deallocation problem in terms of the KID$^-$ interpreter, and in the fifth section we will give an overview of the development of our solution in the rest of this thesis.

2.1 Notation

We will adopt the convention of using double brackets, $[\![\ e\ ]\!]$, around program text. Environments will be represented by ρ, looking up variable x in environment ρ will be represented by $\rho[x]$, and binding variable x to value v in environment ρ by $\rho[v/x]$. Stores will be represented by σ, dereferencing a location l in store σ will be written as $\sigma[l]$, and binding location l to value v in store σ will be written as $\sigma[l \rightarrow v]$. The expression $\mathcal{P}(X)$ indicates the powerset, or set of all subsets, of X. Tuples will be written with angle-brackets and elements separated by commas: $\langle v_1, \cdots, v_n \rangle$. Tagged structures will be written with a subscript tag, $\langle_{Foo}\ v_1, \cdots, v_n \rangle$, and $x.Tag$ will be used to refer to the tag of such a structure. The expression $D_\perp$ constructs a new domain that consists of the elements of domain D plus a new element $\perp$ which is less than all elements of D.

F	$::=$	$\texttt{f}_0 \mid \texttt{f}_1 \mid \ldots$	User Function Names
X	$::=$	$\texttt{x}_0 \mid \texttt{x}_1 \mid \texttt{x}_2 \mid \ldots$	Identifiers
SE	$::=$	$\texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{True} \mid \texttt{False} \mid X$	Simple Expressions
L	$::=$	$l_0 \mid l_1 \mid l_2 \mid \ldots$	Expression Labels
$tag \in Tag$	$=$	$\{1, 2, 3 \cdots\}$	Oneof Tags
OP	$::=$	$+ \mid - \mid \texttt{And} \mid \texttt{Or} \mid \ldots$	Primitive Operators
		$\mid \texttt{MakeTuple} \mid \texttt{Select}_i$	
		$\mid \texttt{MakeArray}_F \mid \texttt{Fetch}$	
		$\mid \texttt{MakeOneof}_{Tag,N} \mid \texttt{Select}_{Tag,N} \mid \texttt{Is?}_{Tag}$	
		$\mid \texttt{Cons} \mid \texttt{Nil} \mid \texttt{Hd} \mid \texttt{Tl} \mid \texttt{Nil?}$	
E	$::=$	$SE \mid {}^{L}OP(SE, \ldots, SE)$	Expressions
		$\mid {}^{L}F(SE, \ldots, SE)$	Function Applications
		$\mid \texttt{if}\,(SE, BE, BE)$	Conditionals
Bs	$::=$	$X = BE; \cdots; X = BE$	Block Bindings
Ds	$::=$	$\texttt{Dealloc}\,(X); \cdots; \texttt{Dealloc}\,(X)$	Deallocation Commands
BE	$::=$	$\{Bs\texttt{---}Ds\ \texttt{in}\ X\} \mid E$	Letrec Blocks
$pr \in Prog$	$::=$	$\{\ldots F(X, \ldots, X) = BE; \ldots;\}$	Programs

Table 2.1: KID$^-$ syntax

We use the adjective *concrete* to refer to values from the standard and instrumented value domains. These are values that arise during actual execution of a program. We use the adjective *abstract* to refer to values that arise during abstract interpretation of a program. These values summarize all the possible values that could arise during the execution of a program under the standard or instrumented interpreters. Hats on values ($\widehat{v}$) or functions ($\widehat{f}$) will be used to denote the abstraction of some value whenever it is not clear from the context that we are talking about an abstract value.

The metalanguage in which the interpreter is written has strict semantics. `Letrec` blocks in the metalanguage are written "`{ x = e in z }`" and have recursive, *i.e.*, `letrec`, scoping rules. The metalanguage can be viewed as a mathematical notation, in which there is no notion of order of evaluation. It can also be viewed as an abstract syntax for a functional language. All of the definitions written in this thesis could be written in a strict, functional language.

2.2 Syntax of KID$^-$

KID$^-$ is intended to be an intermediate language used when compiling Id programs. For this reason, it lacks some features that Id has, such as pattern matching, and so KID$^-$ programs can be rather verbose. KID$^-$ does not have loop expressions — in this work, we interpret and analyze them by translating them into tail recursive functions.

KID$^-$ is a sugared form of the lambda calculus. Functions are named, and a program consists of a top-level recursive block defining the functions in the program. This allows a concise expression of simple programs. The recursive scoping of the program block obviates the need for the Y combinator in the language. Expressions are either constants, variables, conditional expressions, or applications of functions or primitive functions. The syntax of KID$^-$ is shown in Table 2.1.

A program consists of a recursively scoped block of function definitions. A program must define a function named f_0 that takes no arguments — this corresponds to the `main` function in a C program. Interpretation of a program begins by invoking this function. Nested functions are not allowed in this language. For a treatment of how to transform a set of nested function definitions to a flat set of function definitions, see [25] for a description of *lambda lifting*. Also, no currying, or partial application, of functions is supported. Hochheiser [21] describes the compilation of a language with currying into a KID-like intermediate form with no currying. Higher-order functions and closures will be discussed in Section 9.1.

Because KID$^-$ is a first-order language, identifiers are separated into function identifiers F and value identifiers X. Simple expressions are either constants or identifiers. Expressions can be simple expressions, primitive operator applications, function applications, conditional or block expressions. Primitive and user function applications are labeled with a static label drawn from domain L, the set of static expression labels. This expression label will be used in the interpreter to identify objects and procedure activations.

KID$^-$ expressions are divided into two major categories: simple expressions (SE) and expressions (E). The division simplifies many of the clauses of the interpreter, because simple expressions cannot modify or reference the store; they can only reference the environment. All expressions, except block and conditional expressions, consist of an operator and a number of simple expression parameters. In these expressions, each of the parameters can be evaluated by the simple expression evaluator, which does not take or return a store, thus reducing the number of stores that are defined. This reduces the clutter in the evaluator definition. Use of SE is even more pronounced in the instrumented and abstracted interpreters, where more values are returned by the expression evaluator.

Block expressions consist of a set of recursively scoped bindings, a synchronization barrier, and a set of deallocation statements. The interpretation of block expressions is rather involved because KID$^-$ is non-strict and because the scoping of variables in blocks is recursive, *i.e.*, block expressions are `letrec` blocks. The result of a block expression is the value of the final identifier x in the block's inner environment and the block's inner store. The return value may be returned as soon as it is available — block expressions are non-strict and the result value is unaffected by the synchronization barrier. After all computation in the bindings has terminated, each of the deallocation statements is executed — the deallocation statements are *hyperstrict* in each of the bound variables.

Anyplace a block expression is expected, a single expression may be used instead. This allows expressions such as

$$\{\ \mathtt{x} = e; \\ \text{in}\ \mathtt{x}\ \}$$

to be written simply as e whenever `x` is not a free variable of expression e.

The predicate of a conditional is a simple expression, but both branches must be block expressions. Also, the bodies of function definitions must be block expressions. This formulation of the syntax ensures that every structure that is allocated is initially bound to an identifier, because the only place that a structure allocation primitive can occur is on the right-hand side of a block binding.

KID$^-$ has primitives for constructing and manipulating three types of aggregate objects. The primitive `MakeTuple` takes n arguments and constructs an n-tuple from their values. The primitive $\mathtt{Select}_i$ takes a tuple and returns the ith component. The primitive $\mathtt{MakeArray}_F$ takes a length parameter n and some additional arguments, and constructs an array of n elements where each element of the array is the value of function F applied to the index and the additional argument values. `Fetch` takes an array and an index and returns the corresponding element of the array.

$n \in N$	$=$	$\{1, 2, 3 \cdots\}$	Integers
$b \in B$	$=$	$\texttt{True} + \texttt{False}$	Booleans
$l \in L$	$=$	$\{l_0, l_1, l_2, \cdots\}$	Expression Labels
$\alpha \in AL$	$=$	$\epsilon + AL.L$	Activation Labels
$ol \in OL$	$=$	$AL : L$	Object Labels
$v \in V$	$=$	$(N + B + OL)_\perp$	Denotable Values
$tuple \in Tuple$	$=$	$\langle_{Tuple}\ V, \cdots, V\rangle$	Tuples
$v_{array} \in Array$	$=$	$\left\langle_{Array}\ n, \underbrace{V, \cdots, V}_{n}\right\rangle$	Arrays
$v_{oneof} \in Oneof$	$=$	$\langle_{N,N}\ V, \cdots, V\rangle$	Oneofs
$v_{list} \in List$	$=$	$\langle_{Cons}\ V, OL\rangle + \langle_{Nil}\ \rangle$	Lists
$sv \in SV$	$=$	$(Tuple + Array + Oneof + List)_\perp$	Storable Values
$\sigma \in Store$	$=$	$OL \rightarrow SV$	Stores

Table 2.2: Standard value domains

Algebraic types are tagged sums of types. In KID$^-$ we represent algebraic types by oneofs, which are tagged sums of tuples. $\texttt{MakeOneof}_{j,n_{tags}}$ takes m arguments and constructs a oneof tagged with j and m components that belongs to a type with n_{tags} different disjuncts. The tag j of a oneof must be in the range $0 \leq j < n_{tags}$. $\texttt{Select}_{j,i}$ takes a oneof and returns the ith component of that oneof, if the tag of that oneof was j. $\texttt{Is}_j\texttt{?}$ takes a oneof and returns `True` if the tag of that oneof was j.

2.3 KID$^-$ Domains

KID$^-$ has the usual types of values: integers, booleans, tuples, arrays, algebraic types (oneofs), and lists. These value domains are defined in Table 2.2. Figure 2.2 contains the definitions of the least-upper-bound operators on the standard value domains. Figure 2.3 contains the definitions of the ordering operators on the standard value domains. The domains are all naturally ordered.

In order to model sharing of objects properly we will use a store that maps unique labels to tuples. A label unbound in a store will map to $\perp$. Tuples will be passed by reference; We will refer to tuples by their object labels. The actual tuple will reside in an associated store. A denotable value that is a label of an object makes no sense without an associated store. Denotable values, drawn from domain V, are either numbers, booleans, object labels or $\perp$ (undefined).

Object Labels

In order to determine the lifetime of objects, we must be able to distinguish one object from another. Therefore, when objects are created they must be assigned a unique label. We will always refer to the objects by this label.

There are many ways a unique label could be allocated for an object. We could use something like `gensym` to create arbitrary new, unique labels. However, in order to implement the non-strict interpreter, we must be able to deterministically generate a unique label for each instance of each allocation command. We will see later in this chapter that evaluating non-strict, recursive `letrec`

```
{ def f(x,y) =
    { t = l0MakeTuple(x,y)
      result = k0g(t);
     ---
      Dealloc(t);
    in result };

  def g(t) =
    Select1(t)
}
```

Figure 2.1: Labeled deallocation example

blocks involves fixpoint iteration over successive approximations of the recursive environment. Each time we improve the approximation of the environment and store of a `letrec` expression, we must get the same object labels for each object allocated in the block. Therefore the structure of labels must be tied to the structure of the program in some manner so that when we summarize labels we summarize information about particular parts of the program.

In order to name objects uniquely in our interpreter, we will use both a static label from the allocation primitives and a dynamic label identifying the particular invocation of that primitive. Therefore, the domain OL of object labels will consist of two components: a unique activation label and a static label that denotes the expression in the program that allocated the structure.

Static labels are assigned to each expression in a KID$^-$ program. We will only display the pertinent labels on allocation primitives and function applications. These labels will be placed to the left and above the expression that they annotate. In Figure 2.1, we have labeled the `MakeTuple` primitive with l_0 by placing l_0 to the upper left of the `MakeTuple` expression. This label forms the static portion of the object label of any tuple allocated by executing this particular expression.

Activation Labels

We will call the dynamic portion of object labels their *activation labels*. Activation labels are drawn from the domain AL, whose structure will be described below. Our scheme for labeling activations is similar to that of Harrison.

In [19], Harrison uses a pair consisting of a variable name and a procedure string to uniquely name variable instances. In his system, every function expression (lambda abstraction) is uniquely named statically, *e.g.*, λ^{α_0}. The language he is modeling has `call-cc` and this must be reflected in procedure strings, which name a sequential execution path through a program. A procedure string consists of a sequence of lambda names with a superscript of d or u to indicate the entrance or exit from an instance of that procedure, *e.g.*, the procedure string $\alpha_0^d\alpha_1^d\alpha_1^u$ indicates entering the body of lambda α_0, entering the body of lambda expression α_1 followed by exiting the body of lambda expression α_1. Harrison shows that these labels uniquely name every instance of every object allocated in the program.

Harrison's scheme works well in a sequential language in which there is only a single thread of control that can be named by the procedure string. However, in a parallel language such as KID$^-$, there is no sequential thread of control that can uniquely identify each object. Therefore, we will use a variation of procedure strings based on the hierarchical rather than sequential ordering of procedure activations. In this scheme, every expression within a function definition will be uniquely labeled. An instantiation of a procedure f called from a procedure g executing in activation α can be uniquely named by α followed by k_j, where k_j is the label of the expression within procedure g which calls procedure f. Thus, an activation label is the concatenation of the names of the edges in the run-time call-tree, where each edge is labeled by the application expression that created that edge. An important feature of these labels is that the label assigned to a particular instantiation of a procedure invocation will always be the same regardless of the execution order of subexpressions. Since object labels consist of an activation label paired with the expression label of the allocation primitive expression, this feature carries over to object labels.

Our activation labels uniquely identify a particular invocation of a procedure during the execution of a program. Activation labels AL denote a path down the call tree of a program. Activation labels consist of a string of expression labels:

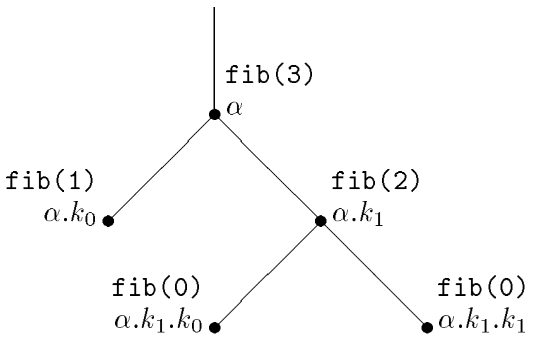
$$\epsilon.k_1.\cdots.k_m$$

where ϵ is the empty activation label and each of the k_i is the expression label of a user function application expression. We use strings of expression labels instead of strings of function names because we must be able to distinguish two invocations of a single procedure within a given procedure activation. Activation label ϵ is used as the activation label of the main body of the program. Each time a procedure is called from activation α, we construct a new activation label by concatenating the expression label k of the function application to α with a ".", yielding $\alpha.k$.

For example, consider the definition of procedure `fib`, shown below. Procedure `fib` is recursive; it contains two calls to itself within the body.

```
def fib(i) =
  { p = i < 2;
    r = if p then 1
        else { n1 = k2fib(i-1);
               n2 = k1fib(i-2);
               n3 = n1 + n2
              in n3 }
   in r };
```

If we invoke $fib(3)$ in activation α, then we get the following activation tree:



$$
\begin{aligned}
v \sqcup \bot &= v \\
\bot \sqcup v &= v \\
v \sqcup \top &= \top \\
\top \sqcup v &= \top
\end{aligned}
$$

$$
v_1 \sqcup_V v_2 = \begin{cases} v_1 & \text{if } v_1 = v_2 \\ \top & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
&\langle_{\mathit{Tuple}}\ v_1, \cdots, v_{n_1}\rangle \sqcup_{SV} \langle_{\mathit{Tuple}}\ w_1, \cdots, w_{n_2}\rangle = \\
&\qquad \begin{cases} \langle_{\mathit{Tuple}}\ (v_1 \sqcup_V w_1), \cdots, (v_{n_1} \sqcup_V w_{n_2})\rangle & \text{if } n_1 = n_2 \\ \top & \text{otherwise} \end{cases} \\
&\langle_{\mathit{Array}}\ n_1, v_1, \cdots, v_{n_1}\rangle \sqcup_{SV} \langle_{\mathit{Array}}\ n_2, w_1, \cdots, w_{n_2}\rangle = \\
&\qquad \begin{cases} \langle_{\mathit{Array}}\ n_1, (v_1 \sqcup_V w_1), \cdots, (v_{n_1} \sqcup_V w_{n_2})\rangle & \text{if } n_1 = n_2 \\ \top & \text{otherwise} \end{cases} \\
&\langle_{\mathit{tag_1}, m_1}\ v_1, \cdots, v_{n_1}\rangle \sqcup_{SV} \langle_{\mathit{tag_2}, m_2}\ w_1, \cdots, w_{n_2}\rangle = \\
&\qquad \begin{cases} \langle_{\mathit{tag_1}, m_1}\ (v_1 \sqcup_V w_1), \cdots, (v_{n_1} \sqcup_V w_{n_2})\rangle & \text{if } \mathit{tag}_1 = \mathit{tag}_2 \wedge m_1 = m_2 \wedge n_1 = n_2 \\ \top & \text{otherwise} \end{cases} \\
&\langle_{\mathit{Cons}}\ v_1, v_2\rangle \sqcup_{SV} \langle_{\mathit{Cons}}\ w_1, w_2\rangle = \langle_{\mathit{Cons}}\ (v_1 \sqcup_V w_1), (v_2 \sqcup_V w_2)\rangle \\
&\langle_{\mathit{Nil}}\ \rangle \sqcup_{SV} \langle_{\mathit{Nil}}\ \rangle = \langle_{\mathit{Nil}}\ \rangle
\end{aligned}
$$

$$
\sigma_1 \sqcup_{\mathit{Store}} \sigma_2 = \lambda ol. \begin{cases} \sigma_1[ol] \sqcup_{SV} \sigma_2[ol] & \text{if } ol \in OL \\ \bot & \text{otherwise} \end{cases}
$$

Figure 2.2: Least upper bound operators on standard value domains

Note that the activations labeled $\alpha.k_0$, $\alpha.k_1$, $\alpha.k_1.k_0$ and $\alpha.k_1.k_1$ can all proceed in parallel with the parent. The only information we get from the activation labels is that parent activations are initiated before their child activations and that parent activations terminate after their child activations terminate.

2.4 KID$^-$ Interpreter

In this section we give an operational semantics for KID$^-$ in terms of a standard interpreter. First, we discuss the evaluation strategy used by the interpreter, then we present the overall structure of the interpreter, then we present the program evaluator, simple expression evaluator, and expression evaluator. Next we discuss the correctness of the interpreter. Finally, we discuss the deallocation problem and give an overview of our solution.

$$\begin{array}{ccc} \bot & \sqsubseteq & v \quad \forall v \\ v & \sqsubseteq & \top \quad \forall v \end{array}$$

$$v_1 \sqsubseteq_V v_2 \quad \equiv \quad \begin{cases} True & \text{if } v_1 = \bot \\ True & \text{if } v_1 = v_2 \\ False & \text{otherwise} \end{cases}$$

$$\sigma_1 \sqsubseteq_{Store} \sigma_2 \quad \equiv \quad \bigwedge_{l_i \in L} \sigma_1[l_i] \sqsubseteq_{SV} \sigma_2[l_i]$$

$$\langle_{Tuple}\ v_1, \cdots, v_{n_1}\rangle \sqsubseteq_{SV} \langle_{Tuple}\ w_1, \cdots, w_{n_2}\rangle \equiv$$
$$\begin{cases} \bigwedge_i (v_i \sqsubseteq_V w_i) & \text{if } n_1 = n_2 \\ False & \text{otherwise} \end{cases}$$
$$\langle_{Array}\ n_1, v_0, \cdots, v_{n_1-1}\rangle \sqsubseteq_{SV} \langle_{Array}\ n_2, w_0, \cdots, w_{n_2-1}\rangle \equiv$$
$$\begin{cases} \bigwedge_{0 \le i < n_1} (v_i \sqsubseteq_V w_i) & \text{if } n_1 = n_2 \\ False & \text{otherwise} \end{cases}$$
$$\langle_{tag_1, m_1}\ v_1, \cdots, v_{n_1}\rangle \sqsubseteq_{SV} \langle_{tag_2, m_2}\ w_1, \cdots, w_{n_2}\rangle \equiv$$
$$\begin{cases} \bigwedge_i (v_i \sqsubseteq_V w_i) & \text{if } tag_1 = tag_2 \wedge m_1 = m_2 \wedge n_1 = n_2 \\ False & \text{otherwise} \end{cases}$$
$$\langle_{Cons}\ v_1, v_1\rangle \sqsubseteq_{SV} \langle_{Cons}\ w_1, w_2\rangle \equiv$$
$$(v_1 \sqsubseteq_V w_1) \wedge (v_2 \sqsubseteq_V w_2)$$
$$\langle_{Nil}\ \rangle \sqsubseteq_{SV} \langle_{Nil}\ \rangle \equiv True$$

Figure 2.3: Ordering operators on standard domains

2.4.1 Evaluation Strategy

This interpreter is somewhat novel in that it evaluates each expression more than once in order to implement the non-strictness of the KID$^-$ language. Typically, an interpreter will evaluate each expression exactly once.

Consider the following KID$^-$ fragment, which uses non-strictness to define the second component of the tuple in terms of the first component of the tuple.

```
{ a = l0MakeTuple(x,y);
  x = 2;
  y = Select1(a);
 in a }
```

There is no order in which we can evaluate the three bindings of this expression in order to completely specify the expression. The evaluation strategy used by this interpreter is to repeatedly evaluate subexpressions in successively improved environments and stores until a limit is reached and the expression is fully evaluated. The interpreter will first approximate the environment of the body of the block expression by creating and environment in which each of the bound variables is bound to $\perp$. Then it will evaluate each of the right-hand-side expressions in that environment to yield new approximations to the values of the bound variables and new approximations to the value of the store. This process is repeated until both the environment and the store have stabilized.

For this example, the interpreter would start with environment ρ^0 and store σ^0:

$$\begin{aligned} \rho^0 &= \begin{bmatrix} \mathtt{a} \rightarrow \perp \\ \mathtt{x} \rightarrow \perp \\ \mathtt{y} \rightarrow \perp \end{bmatrix} \\ \sigma^0 &= \perp_{Store} \end{aligned}$$

After evaluating each of the right-hand-sides in environment ρ_0 and store σ_0 and combining the results into a new environment and store, we get ρ^1 and σ^1:

$$\begin{aligned} \rho^1 &= \begin{bmatrix} \mathtt{a} \rightarrow \alpha : l_0 \\ \mathtt{x} \rightarrow \underline{2} \\ \mathtt{y} \rightarrow \perp \end{bmatrix} \\ \sigma^1 &= \begin{bmatrix} \alpha : l_0 \rightarrow \langle_{\,Tuple}\ \perp, \perp \rangle \end{bmatrix} \end{aligned}$$

in which variables `a` and `x` have non-bottom bindings and label $\alpha : l_0$ is bound to a tuple containing $\perp$ and $\perp$.

One more iteration would yield ρ^2 and σ^2:

$$\begin{aligned} \rho^2 &= \begin{bmatrix} \mathtt{a} \rightarrow \alpha : l_0 \\ \mathtt{x} \rightarrow \underline{2} \\ \mathtt{y} \rightarrow \perp \end{bmatrix} \\ \sigma^2 &= \begin{bmatrix} \alpha : l_0 \rightarrow \langle_{\,Tuple}\ \underline{2}, \perp \rangle \end{bmatrix} \end{aligned}$$

Now the first component of the tuple labeled $\alpha : l_0$ contains $\underline{2}$.

Yet one more iteration would yield ρ^3 and σ^3:

$$\begin{aligned} \rho^3 &= \begin{bmatrix} \mathtt{a} \rightarrow \alpha : l_0 \\ \mathtt{x} \rightarrow \underline{2} \\ \mathtt{y} \rightarrow \underline{2} \end{bmatrix} \\ \sigma^3 &= \begin{bmatrix} \alpha : l_0 \rightarrow \langle_{\,Tuple}\ \underline{2}, \perp \rangle \end{bmatrix} \end{aligned}$$

in which variable `y` is now bound to $\underline{2}$.

Finally, we would reach the environment ρ^4 and store σ^4 of the completely evaluated block expression:

$$\begin{aligned} \rho^4 &= \begin{bmatrix} \mathtt{a} \rightarrow \alpha : l_0 \\ \mathtt{x} \rightarrow \underline{2} \\ \mathtt{y} \rightarrow \underline{2} \end{bmatrix} \\ \sigma^4 &= \begin{bmatrix} \alpha : l_0 \rightarrow \langle_{\,Tuple}\ \underline{2}, \underline{2} \rangle \end{bmatrix} \end{aligned}$$

in which all three variables have non-bottom bindings and the tuple has no bottom components.

We can tell that environment ρ^4 and store σ^4 have reached the fixpoint by iterating the process one more time. This iteration yields the same result as the previous iteration; therefore, ρ^4 and σ^4 must be the complete value.

$$\begin{aligned}
\rho^5 &= \begin{bmatrix} \mathtt{a} \rightarrow \alpha : l_0 \\ \mathtt{x} \rightarrow \underline{2} \\ \mathtt{y} \rightarrow \underline{2} \end{bmatrix} \\
\sigma^5 &= \begin{bmatrix} \alpha : l_0 \rightarrow \langle_{Tuple}\ \underline{2}, \underline{2} \rangle \end{bmatrix}
\end{aligned}$$

The important thing to notice is that each expression was evaluated five times in order to reach the fixpoint. The evaluation strategy we have chosen had some effect on how we named objects. We had to be able to deterministically assign a label to an object in order to evaluate `MakeTuple` expressions multiple times.

Most interpreters for non-strict languages use a rewrite system, where subexpressions are rewritten when they are evaluated. We chose this evaluation strategy because when we abstract the interpreter we want the compiler to analyze programs by recursively descending the program, evaluating as it goes along. We want the program being evaluated to have the same text as the program being annotated or verified; so we do not want to use a rewriting interpreter.

Arrays

Here is an example of the use of `MakeArray`.

```
{
 def g1 (i, x, y) =
   l0MakeTuple(x,y,i);

 def f1 (n, x, y) =
   l1MakeArrayg1(n, x, y);
}
```

This example consists of two procedures. Procedure `g1`, which takes three values, allocates a three-tuple containing the three values and returns the value as its result. Procedure `f1` uses `MakeArray` to construct an n-element array with v_i as the ith element of the matrix:

$$v_i = \mathtt{g1}(i, x, y)$$

where $(0 \leq i < n)$.

The value v and store σ resulting from a call to procedure `f1` with values 3, 17 and 22 in activation α would be:

$$\begin{aligned}
v &= \alpha : l_1 \\
\sigma &= \begin{bmatrix} \alpha : l_1 \rightarrow \langle_{Array}\ 3, \alpha.l_1.0 : l_0, \alpha.l_1.1 : l_0, \alpha.l_1.2 : l_0 \rangle \\ \alpha.l_1.0 : l_0 \rightarrow \langle_{Tuple}\ 22, 23, 0 \rangle \\ \alpha.l_1.1 : l_0 \rightarrow \langle_{Tuple}\ 22, 23, 1 \rangle \\ \alpha.l_1.2 : l_0 \rightarrow \langle_{Tuple}\ 22, 23, 2 \rangle \end{bmatrix}
\end{aligned}$$

Please note that procedure `g1` is invoked with the index i of the array element as well as the values of the two extra parameters passed to `MakeArray`. Any number of additional parameters may be passed to the element creation function through the extra parameters to `MakeArray`. These extra parameters increase the expressiveness of the language without having higher-order functions.

Algebraic Types

In some cases, we would like to represent a value whose type is one of a number of different types. In this case, we use an algebraic type, which is a disjoint union of the types. We will refer to algebraically typed objects as oneofs. In order to maintain type safety, a disjunct tag is maintained on these objects, and special operators are provided to construct and manipulate them.

For example, consider the `transaction` algebraic type defined below.

```
type transaction = deposit I | withdrawal I
```

A transaction is either a deposit or a withdrawal.

Transactions are represented by tagged structures in the standard semantics. A transaction object is either a deposit:

$$\langle_{0,2}\ n\rangle$$

where the subscript $0, 2$ indicates the 0th disjunct of a type with two disjuncts, or a withdrawal:

$$\langle_{1,2}\ m\rangle\,.$$

where the subscript $1, 2$ indicates the 1th disjunct of a type with two disjuncts. Any particular transaction value will be either a deposit or a withdrawal.

The KID$^-$ code to create and manipulate structures of the `transaction` type using these primitives would look like:

```
def make_deposit(n) =
  ^l0 MakeOneof_{0,2}(n);

def make_withdrawal(m) =
  ^l0 MakeOneof_{1,2}(m);

def deposit_amount(d) =
  if Is_0(d)
    then Select_{0,1}(d);
    else Error();

def withdrawal_amount(w) =
  if Is_1(w)
     then Select_{1,1}(w);
     else Error();

def deposit?(t) =
  Is_0(t);
```

where `Error` is a primitive that always returns bottom (and presumably drops the user into the debugger).

Some algebraic types are defined recursively. These types are used to represent things such as lists, trees, and graphs. Because lists are used so often in functional programs, KID$^-$ has primitives specifically defined to create and manipulate list objects.

2.4.2 Interpreter Structure

This section introduces the structure of the standard interpreter and defines several properties that the interpreter satisfies. The interpreter consists of three semantic functions, $\mathcal{SE}$, $\mathcal{E}$ and $\mathcal{PE}$, which together interpret KID$^-$ programs. The following are the signatures of the semantic functions that make up the interpreter.

$\mathcal{SE}$	:	$SE \rightarrow Env \rightarrow V$	Evaluates simple expressions
$\mathcal{E}$	:	$E \rightarrow Env \rightarrow Store \rightarrow AL \rightarrow (V \times Store)$	Evaluates expressions
$\mathcal{PE}$	:	$Prog \rightarrow (V \times Store)$	Evaluates programs

where Env, the domain of environments, is defined as:

$$Env \quad = \quad X \rightarrow V.$$

Environments map identifiers, or variables, to values. An identifier that is unbound in an environment maps to $\perp$.

The function $\mathcal{PE}$ evaluates a program and returns a denotable value and a store as the result. The function $\mathcal{E}$ takes an expression e, an environment ρ, a store σ, and an activation label α and returns the denotable value and new store resulting from evaluating the expression e in ρ, σ, and α. We call a triple consisting of an environment, a store, and an activation label a *context* — it contains the contextual information necessary to interpret an expression.

Definition 2.1 (Dynamic Context) *A* dynamic context *is a triple consisting of an environment, a store, and an activation label.*

In order to show that the interpreter is sound, *i.e.*, it terminates on well-behaved (non-looping) programs, we must show that procedures $\mathcal{SE}$ and $\mathcal{E}$ are monotonic. Monotonicity is required to show the existence of the fixpoints computed during evaluation of `letrec` blocks. The monotonicity of $\mathcal{E}$ depends on a property called *extensionality.*

Definition 2.2 (Extensionality) *A function f is* extensive *if,*

$$\forall x \in Domain(f).\ x \sqsubseteq f(x)$$

In other words, the result of $f(x)$ always includes x — function f only adds information to its argument.

We will have to show that $\mathcal{E}$ is extensive, that is, $\mathcal{E}$ only adds to the bindings of the store that it takes as input. The set of locations bound in the store resulting from a call to evaluator $\mathcal{E}$ will be a superset of the set of locations bound in the input store.

Proposition 2.3 ($\mathcal{E}$ is extensive)

$$\begin{array}{l} \forall \rho \in Env, \sigma_0 \in Store, \alpha \in AL, \\ \exists \ \langle \ v, \sigma_1 \ \rangle = \mathcal{E} [\![\ e \ ]\!] \, \rho \, \sigma_0 \, \alpha \ , \\ \Rightarrow \sigma_0 \sqsubseteq \sigma_1 \end{array}$$

The extensionality of $\mathcal{E}$ is used in the proof of the monotonicity of $\mathcal{E}$.

Proposition 2.4 ($\mathcal{E}$ is monotonic)

$$\begin{array}{l} \forall \rho_0, \rho_1 \in Env, \sigma_0, \sigma_1 \in Store, \alpha \in AL, \\ \exists \ \langle \ v_0, \sigma_0' \ \rangle = \mathcal{E} [\![\ e \ ]\!] \, \rho_0 \, \sigma_0 \, \alpha \ , \\ \exists \ \langle \ v_1, \sigma_1' \ \rangle = \mathcal{E} [\![\ e \ ]\!] \, \rho_1 \, \sigma_1 \, \alpha \ , \\ (\rho_0 \sqsubseteq \rho_1) \wedge (\sigma_0 \sqsubseteq \sigma_1) \Rightarrow (v_0 \sqsubseteq v_1) \wedge (\sigma_0' \sqsubseteq \sigma_1') \end{array}$$

2.4.3 KID$^-$ Program Evaluator

The program evaluator $\mathcal{PE}$ evaluates the main function by invoking the expression evaluator with the text of the body of the function, an empty environment, an empty store, and an empty activation label. The definition of the program evaluator is given below:

$$\begin{array}{l} \mathcal{PE} [\![\ pr \ ]\!] \ = \\ \qquad \{ \ \{ \cdots f_i(x_{i,1}, \ldots, x_{i,n_i}) = e_i; \cdots \} = pr; \\ \qquad \ \ \text{in } \mathcal{E} [\![\ e_0 \ ]\!] \perp_{Env} \perp_{Store} \epsilon \ \} \end{array}$$

where

$$f_0() = e_0$$

is the definition of the main procedure f_0 in program pr.

The purpose of the program evaluator is to provide the initial environment to the expression evaluator so that it may evaluate the body of the program. Function identifiers are handled specially; they are not bound in the environment. A different model of program evaluation would yield an environment of functions, and one could invoke any of the procedures in the program with arbitrary arguments. We chose the whole program view because it is simple and because it is consistent with the approach of many systems where programs are compiled and run as a single unit with a single entry point.

2.4.4 KID$^-$ Simple Expression Evaluator

The simple expression evaluator takes a simple expression and an environment and returns a denotable value. It is used by the expression evaluator. Numeric and boolean literals are evaluated to numeric and boolean constants. Identifiers, or variables, are evaluated by looking them up in the environment.

$$\begin{array}{lcll} \mathcal{SE} [\![\ n \ ]\!] \, \rho & = & \underline{n} & \text{where } n \text{ is a number} \\ \mathcal{SE} [\![\ b \ ]\!] \, \rho & = & \underline{b} & \text{where } b \text{ is a boolean} \\ \mathcal{SE} [\![\ x \ ]\!] \, \rho & = & \rho[x] & \text{where } x \text{ is a variable} \end{array}$$

Simple expressions cannot modify the store, so no store is passed into or returned from procedure $\mathcal{SE}$.

2.4.5 KID$^-$ Expression Evaluator

The expression evaluator will now be defined as a dispatch function on the structure of the input term. Remember that the expression evaluator takes an expression, an environment, a store, and an activation label, and returns a value and a new store.

Evaluation of Simple and Primitive Expressions

The first three clauses of the interpreter define the semantics of constants and variables:

$$\begin{array}{lcll}
\mathcal{E}[\![\ n\ ]\!]\,\rho\,\sigma\,\alpha & = & \langle\, \mathcal{SE}[\![\ n\ ]\!]\,\rho\,,\sigma\,\rangle & \text{where } n \text{ is a number} \\
\mathcal{E}[\![\ b\ ]\!]\,\rho\,\sigma\,\alpha & = & \langle\, \mathcal{SE}[\![\ b\ ]\!]\,\rho\,,\sigma\,\rangle & \text{where } b \text{ is a boolean} \\
\mathcal{E}[\![\ x\ ]\!]\,\rho\,\sigma\,\alpha & = & \langle\, \mathcal{SE}[\![\ x\ ]\!]\,\rho\,,\sigma\,\rangle & \text{where } x \text{ is a variable}
\end{array}$$

All three of these clauses call the simple expression evaluator, and in these clauses the input store is returned unchanged because simple expressions cannot modify the store.

The next clause shows the evaluation of a simple arithmetic expression.

$$\begin{array}{lcl}
\mathcal{E}[\![\ +(se_1, se_n)\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ v_1 = \mathcal{SE}[\![\ se_1\ ]\!]\,\rho\ ; \\
 & & \phantom{\{\ } v_2 = \mathcal{SE}[\![\ se_2\ ]\!]\,\rho \\
 & & \phantom{\{\ } \text{in } \langle\ v_1 + v_2, \sigma\,\rangle\ \}
\end{array}$$

The two operands are evaluated first, then the primitive operator + is applied to those values, and the result is returned. These primitive operators do not modify the store; consequently, it is returned unchanged.

Evaluation of Function Applications

We evaluate an application expression by evaluating its arguments and forming environment ρ' and activation label α'. Environment ρ' is obtained by extending the empty environment with bindings from each of the formal parameters to their actual values. We concatenate activation label α with the expression label k of the activation expression to form the new activation label α'. Then we evaluate the body of the function in environment ρ', store σ, and activation label α'. The non-strictness of functions and data structures is handled by the implementation of `letrec` blocks, shown later in this section. So if we evaluate the body of a procedure before all of its arguments have been evaluated, those arguments will be undefined ($\perp$) or partially defined (if they are bound to labels of data structures).

$$\begin{array}{lcl}
\mathcal{E}[\![\ {}^{k}f(se_1, \cdots, se_n)\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ v_1 = \mathcal{SE}[\![\ se_1\ ]\!]\,\rho\ ; \\
 & & \phantom{\{\ v_1 =\ } \vdots \\
 & & \phantom{\{\ } v_n = \mathcal{SE}[\![\ se_n\ ]\!]\,\rho\ ; \\
 & & \phantom{\{\ } \rho' = \perp_{Env}[v_1/x_1, \cdots, v_n/x_n]; \\
 & & \phantom{\{\ } \alpha' = \alpha.k; \\
 & & \phantom{\{\ } \text{in } \mathcal{E}[\![\ e\ ]\!]\,\rho'\,\sigma\,\alpha'\ \} \\
 & & \\
 & & \text{where } f(x_1, \cdots, x_n) = e \text{ is a definition in the program}
\end{array}$$

Note that the body of function f is evaluated in a new environment and a new activation $\alpha.k$ consisting of the current activation label α concatenated with the expression label k of the application expression.

Evaluation of Conditionals

Conditionals are evaluated by first interpreting the predicate, and then interpreting one of the branches of the conditional depending on the value of the predicate.

$$\begin{array}{lcl} \mathcal{E}[\![\ \mathtt{if}\,(se_0, e_1, e_2) \ ]\!] \rho\,\sigma\,\alpha & = & \mathit{if}\ \mathcal{SE}[\![\ se_0 \ ]\!] \rho \ = \mathsf{True} \\ & & \quad \mathit{then}\, \mathcal{E}[\![\ e_1 \ ]\!] \rho\,\sigma\,\alpha \\ & & \quad \mathit{else}\, \mathcal{E}[\![\ e_2 \ ]\!] \rho\,\sigma\,\alpha \end{array}$$

Evaluation of Block Expressions

Evaluation of KID$^-$ `letrec` blocks is rather complex because they have recursive scope and because KID$^-$ is non-strict. They are evaluated by solving the recursive equations resulting from interpreting each of the binding right-hand-sides in an environment that has the `letrec` block variables bound to the values of the binding right-hand-sides. This recursive equation is solved by fixpoint iteration of function $EvalBindings$, starting with an initial approximation of the environment that binds each of the x_i to bottom and an initial approximation of the store equal to the incoming store.

After the bindings have been evaluated completely, the deallocation statements are executed. The deallocation statements have no effect in the standard interpreter, but they will be modeled more precisely in the instrumented interpreter in Chapter 3.

$$\begin{array}{l} \mathcal{E}[\![\ \{\ Bs\texttt{---}Ds\ \mathtt{in}\ x\} \ ]\!] \rho\,\sigma\,\alpha \ = \\ \quad \begin{array}{ll} \{\ [\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!] & = Bs; \\ \ \ [\![\ \mathtt{Dealloc}\,(y_1); \cdots; \mathtt{Dealloc}\,(y_k) \ ]\!] & = Ds; \\ \ \ \rho_0 & = \rho[\perp/x_1, \cdots, \perp/x_n]; \\ \ \ \langle\ \rho', \sigma' \ \rangle & = \mathit{EvalBindings}\,(Bs, \rho_0, \sigma, \alpha); \\ \ \ \text{in}\ \langle\ \rho'[x], \sigma' \ \rangle\ \} & \end{array} \end{array}$$

where

$$\begin{array}{l} \mathit{EvalBindings}\,([\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!] , \rho, \sigma, \alpha) = \\ \quad \begin{array}{lll} \{\ \langle\ v_1, \sigma_1 \ \rangle & = \mathcal{E}[\![\ e_1 \ ]\!] \rho\,\sigma\,\alpha\,; & \\ & \ \vdots & \\ \ \ \langle\ v_n, \sigma_n \ \rangle & = \mathcal{E}[\![\ e_n \ ]\!] \rho\,\sigma\,\alpha\,; & \\ \ \ \rho' & = \rho[(v_1 \sqcup \rho[x_1])/x_1, \cdots, (v_n \sqcup \rho[x_n])/x_n]; & \\ \ \ \sigma' & = \bigsqcup_i \sigma_i; & \\ \ \ \langle\ \rho'', \sigma'' \ \rangle & = \ \mathit{if}\ \rho' = \rho \wedge \sigma' = \sigma & \\ & \qquad \mathit{then}\ \langle\ \rho', \sigma' \ \rangle & \\ & \qquad \mathit{else}\ \mathit{EvalBindings}\,([\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!] , \rho', \sigma', \alpha) & \\ \ \ \text{in}\ \langle\ \rho'', \sigma'' \ \rangle\ \} & & \end{array} \end{array}$$

Evaluation of Tuple Primitives

The next two clauses give the evaluation rules for tuple data structures. The primitive `MakeTuple` takes m values and returns a structure containing those m values. This clause constructs a unique object label *ol* by pairing the current activation label and the expression label of the `MakeTuple` expression, and returns a store that has *ol* bound to the new tuple in the incoming store. The object label is returned as the value of the expression. This clause only adds information to the store, thus preserving the extensionality of $\mathcal{E}$.

$$\mathcal{E}[\![\ {}^{l}\texttt{MakeTuple}(se_1,\cdots,se_m)\ ]\!]\,\rho\,\sigma\,\alpha \quad = \quad \begin{array}{l} \{\ v_1 \quad = \mathcal{SE}[\![\ se_1\ ]\!]\,\rho\ ; \\ \qquad\quad \vdots \\ \ \ v_n \quad = \mathcal{SE}[\![\ se_m\ ]\!]\,\rho\ ; \\ \ \ ol \quad = \alpha : l; \\ \ \ v_{tuple} = \langle_{\mathit{Tuple}}\ v_1,\cdots,v_m\rangle\ ; \\ \ \ \sigma' \quad = \sigma[ol \rightarrow (v_{tuple} \sqcup \sigma[ol])]; \\ \ \ \text{in}\ \langle\ ol,\sigma'\ \rangle\ \} \end{array}$$

Tuple selection is accomplished by evaluating the argument to the Select_i primitive, yielding an object label, and looking up the value of that object label in the current store. The ith component of that tuple is returned as a value, along with the current store.

$$\mathcal{E}[\![\ \texttt{Select}_i(se)\ ]\!]\,\rho\,\sigma\,\alpha \quad = \quad \begin{array}{l} \{\ ol \qquad\qquad\qquad = \mathcal{SE}[\![\ se\ ]\!]\,\rho\ ; \\ \ \ \langle_{\mathit{Tuple}}\ v_1,\cdots,v_n\rangle = \sigma[ol]; \\ \ \ \text{in}\ \langle\ v_i,\sigma\ \rangle\ \} \end{array}$$

Evaluation of Array Primitives

The following three clauses give the evaluation rules for array data structure operators: `MakeArray`, `Fetch`, and `Bounds`. The primitive $\texttt{MakeArray}_{f_i}$ takes a simple expression that evaluates to length n and r simple expressions that evaluate to values to pass to function f_i, and makes an array of length n where the jth component is $f_i(j,v_1,\cdots,v_r)$. Note that this clause only adds information to the store, thus preserving the extensionality of $\mathcal{E}$.

$$\begin{array}{l} \mathcal{E}[\![\ {}^{k}\texttt{MakeArray}_{f_i}(se_0,se_1,\cdots,se_r)\ ]\!]\,\rho\,\sigma\,\alpha\ = \\ \quad \begin{array}{ll} \{\ ol & = \alpha : k; \\ \ \ n & = \mathcal{SE}[\![\ se_0\ ]\!]\,\rho\ ; \\ \ \ v_1 & = \mathcal{SE}[\![\ se_1\ ]\!]\,\rho\ ; \\ & \ \vdots \\ \ \ v_r & = \mathcal{SE}[\![\ se_r\ ]\!]\,\rho\ ; \\ \ \ \langle\ u_0,\sigma_0\ \rangle & = \mathcal{E}[\![\ e_i\ ]\!]\,(\perp_{Env}[\underline{0}/x_0,v_1/x_1,\cdots,v_r/x_r])\,\sigma\,(\alpha.k.0)\ ; \\ & \ \vdots \\ \ \ \langle\ u_{n-1},\sigma_{n-1}\ \rangle & = \mathcal{E}[\![\ e_i\ ]\!]\,(\perp_{Env}[\underline{n-1}/x_0,v_1/x_1,\cdots,v_r/x_r])\,\sigma\,(\alpha.k.(n-1))\ ; \\ \ \ v_{array} & = \langle_{\mathit{Array}}\ n,u_0,\cdots,u_{n-1}\rangle\ ; \\ \ \ \sigma' & = \sigma[ol \rightarrow (v_{array} \sqcup \sigma[ol])] \sqcup \left(\bigsqcup\limits_{0\le i<n} \sigma_i\right)\ ; \\ \ \ \text{in}\ \langle\ ol,\sigma'\ \rangle\ \} \end{array} \\ \quad \text{where}\ f_i(x_0,x_1,\cdots,x_r) = e_i\ \text{is a definition in the program.} \end{array}$$

The primitive `Fetch` takes an array a and an index i, and returns the ith component of a.

$$\mathcal{E}[\![\ {}^{k}\texttt{Fetch}(se_1,se_2)\ ]\!]\,\rho\,\sigma\,\alpha\ =$$

$$
\begin{array}{l}
\{\ ol = \mathcal{SE}\,[\![\ se_1\ ]\!]\,\rho\ ; \\
\ \ i\ \ = \mathcal{SE}\,[\![\ se_2\ ]\!]\,\rho\ ; \\
\ \ \langle_{Array}\ n, v_0, \cdots, v_{n-1}\rangle = \sigma[ol]; \\
\ \ \text{in}\ \langle\ v_i, \sigma\ \rangle\ \}
\end{array}
$$

The primitive `Bounds` takes an array and returns the length of the array.

$$
\begin{array}{l}
\mathcal{E}\,[\![\ {}^{k}\texttt{Bounds}\,(se)\ ]\!]\,\rho\,\sigma\,\alpha\ = \\
\qquad \{\ ol = \mathcal{SE}\,[\![\ se\ ]\!]\,\rho\ ; \\
\qquad\ \ \langle_{Array}\ n, v_0, \cdots, v_{n-1}\rangle = \sigma[ol]; \\
\qquad\ \ \text{in}\ \langle\ n, \sigma\ \rangle\ \}
\end{array}
$$

Evaluation of Algebraic Type Primitives

The following three clauses define the behavior of the interpreter on the primitives that allocate oneofs, select components from oneofs, and test the tags of oneofs. The primitive $\texttt{MakeOneof}_{tag,n_{tags}}$ allocates a oneof whose tag is tag and which belongs to a type with n_{tags} tags and whose elements are the values of simple expressions se_1 through se_m. The primitive $\texttt{Is}_{tag}\texttt{?}$ returns `True` if the tag of the oneof to which simple expression se evaluates is tag. The primitive $\texttt{Select}_{tag,i}$ returns the ith component of the oneof to which se evaluates if the tag of that object is tag; otherwise it returns $\perp$.

$$
\begin{array}{l}
\mathcal{E}\,[\![\ {}^{l}\texttt{MakeOneof}_{tag,n_{tags}}\,(se_1, \cdots, se_m)\ ]\!]\,\rho\,\sigma\,\alpha\ = \\
\qquad \{\ v_1 \qquad = \mathcal{SE}\,[\![\ se_1\ ]\!]\,\rho\ ; \\
\qquad\qquad\qquad \vdots \\
\qquad\ \ v_{n_i} \qquad = \mathcal{SE}\,[\![\ se_{n_i}\ ]\!]\,\rho\ ; \\
\qquad\ \ ol \qquad = \alpha : l; \\
\qquad\ \ v_{oneof} = \left\langle_{i,n_{tags}}\ v_1, \cdots, v_m\right\rangle; \\
\qquad\ \ \sigma' \qquad = \sigma[ol \rightarrow (v_{oneof} \sqcup \sigma[ol])]; \\
\qquad\ \ \text{in}\ \langle\ ol, \sigma'\ \rangle\ \} \\
\mathcal{E}\,[\![\ \texttt{Is}_{tag}\texttt{?}\,(se)\ ]\!]\,\rho\,\sigma\,\alpha\ = \\
\qquad \{\ ol = \mathcal{SE}\,[\![\ se\ ]\!]\,\rho\ ; \\
\qquad\ \ \left\langle_{tag',n_{tags}}\ v_0, \cdots, v_m\right\rangle = \sigma[ol]; \\
\qquad\ \ b\ =\ \ \textit{if } tag = tag' \\
\qquad\qquad\qquad \textit{then}\ \mathsf{True} \\
\qquad\qquad\qquad \textit{else}\ \mathsf{False}; \\
\qquad\ \ \text{in}\ \langle\ b, \sigma\ \rangle\ \} \\
\mathcal{E}\,[\![\ \texttt{Select}_{tag,i}\,(se)\ ]\!]\,\rho\,\sigma\,\alpha\ = \\
\qquad \{\ ol = \mathcal{SE}\,[\![\ se\ ]\!]\,\rho\ ; \\
\qquad\ \ \left\langle_{tag',n_{tags}}\ v_1, \cdots, v_m\right\rangle = \sigma[ol]; \\
\qquad\ \ v =\ \ \textit{if } tag = tag'\ ; \\
\qquad\qquad\quad \textit{then}\ v_i \\
\qquad\qquad\quad \textit{else}\ \perp \\
\qquad\ \ \text{in}\ \langle\ v, \sigma\ \rangle\ \}
\end{array}
$$

Evaluation of List Primitives

The following five clauses give the semantics of the list manipulation primitives. The primitive constructor `Cons` takes an element x and a list v_{list} and constructs a new list with x as its head and v_{list} as its tail. The primitives `Hd` and `Tl` take a list and return the head and tail, respectively, of the list. The constructor `Nil` returns a new empty list. The predicate `Nil?` returns True if the value is `Nil` and `False` otherwise.

$$
\begin{array}{rcl}
\mathcal{E}[\![\ {}^{l}\texttt{Cons}\,(se_1, se_2)\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ v_1 \ = \mathcal{SE}[\![\ se_1\ ]\!]\,\rho\ ; \\
 & & \quad v_2 \ = \mathcal{SE}[\![\ se_2\ ]\!]\,\rho\ ; \\
 & & \quad ol \ = \alpha : l; \\
 & & \quad v_{cons} = \langle_{Cons}\ v_1, v_2 \rangle\ ; \\
 & & \quad \sigma' \ = \sigma[ol \rightarrow (v_{cons} \sqcup \sigma[ol])]; \\
 & & \quad \text{in}\ \langle\ ol, \sigma'\ \rangle\ \} \\
\mathcal{E}[\![\ \texttt{Hd}\,(se)\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ ol = \mathcal{SE}[\![\ se\ ]\!]\,\rho\ ; \\
 & & \quad \langle_{Cons}\ v_1, v_2 \rangle = \sigma[ol]; \\
 & & \quad \text{in}\ \langle\ v_1, \sigma\ \rangle\ \} \\
\mathcal{E}[\![\ \texttt{Tl}\,(se)\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ ol = \mathcal{SE}[\![\ se\ ]\!]\,\rho\ ; \\
 & & \quad \langle_{Cons}\ v_1, v_2 \rangle = \sigma[ol]; \\
 & & \quad \text{in}\ \langle\ v_2, \sigma\ \rangle\ \} \\
\mathcal{E}[\![\ {}^{l}\texttt{Nil}\,()\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ ol = \alpha : l; \\
 & & \quad \sigma' = \sigma[ol \rightarrow (\langle_{Nil}\ \rangle \sqcup \sigma[ol])]; \\
 & & \quad \text{in}\ \langle\ ol, \sigma'\ \rangle\ \} \\
\mathcal{E}[\![\ \texttt{Nil?}\,(se)\ ]\!]\,\rho\,\sigma\,\alpha & = & \{\ ol = \mathcal{SE}[\![\ se\ ]\!]\,\rho\ ; \\
 & & \quad b \ = \ \textit{if}\ \sigma[ol].\texttt{tag} = \texttt{Nil} \\
 & & \qquad\qquad \textit{then}\ \mathsf{True} \\
 & & \qquad\qquad \textit{else}\ \mathsf{False}; \\
 & & \quad \text{in}\ \langle\ b, \sigma\ \rangle\ \}
\end{array}
$$

2.4.6 Soundness of Standard Interpreter

Theorem 2.3 *The interpreter $\mathcal{E}$ is extensive with respect to the store.*

$$
\begin{array}{l}
\forall \rho \in Env, \sigma_0 \in Store, \alpha \in AL, \\
\exists v \in V, \sigma_1 \in Store \\
\langle\ v, \sigma_1\ \rangle = \mathcal{E}[\![\ e\ ]\!]\,\rho\,\sigma_0\,\alpha\ \Rightarrow\ \sigma_0 \sqsubseteq \sigma_1
\end{array}
$$

Proof:

By structural induction:

- The clauses that interpret simple expressions and arithmetic primitives return the store unchanged, and so these clauses are extensive with respect to the store.
- The clauses that interpret conditional and function application expressions call the interpreter on their subexpressions with their input store. Assuming that interpretation of subexpressions (the inductive case) is extensive, then the interpretation of conditional and function application expressions is extensive with respect to the store.

- To prove the extensionality of the clause that interprets `letrec` blocks, we have to show that *EvalBindings* is extensive with respect to the store. This function computes the solution to the set of recursive equations formed by the block bindings. The solution consists of an environment ρ' and a store σ'. The store σ' must include the input store σ, because *EvalBindings* calls the interpreter on the binding right-hand sides with the input store σ, and then takes the least upper bound of the resulting stores. Both of these steps are extensive.
- Each of the clauses that allocate structures are extensive because they only add a binding to the store.
- Each of the clauses that fetch values from structures are extensive because they return the input store unchanged.

■

Theorem 2.4 *The interpreter functions $\mathcal{SE}$ and $\mathcal{E}$ are monotonic. Given simple expression se, expression e, and activation label α, we show monotonicity with respect to the environment and store:*

$$\forall \rho_0, \rho_1 \in \mathit{Env}, \sigma_0, \sigma_1 \in \mathit{Store},$$
$$\rho_0 \sqsubseteq \rho_1 \Rightarrow \mathcal{SE}[\![\ se\ ]\!]\,\rho_0 \ \sqsubseteq \mathcal{SE}[\![\ se\ ]\!]\,\rho_1$$

$$\rho_0 \sqsubseteq \rho_1 \wedge \sigma_0 \sqsubseteq \sigma_1 \Rightarrow \mathcal{E}[\![\ e\ ]\!]\,\rho_0\,\sigma_0\,\alpha \ \sqsubseteq \mathcal{E}[\![\ e\ ]\!]\,\rho_1\,\sigma_1\,\alpha$$

Proof:

First $\mathcal{SE}$, by structural induction:

- The clauses that evaluate numeric and boolean literals always return the values of those literals; therefore, $\mathcal{SE}[\![\ c\ ]\!]\,\rho_0 \ \sqsubseteq \ \mathcal{SE}[\![\ c\ ]\!]\,\rho_1$, because the values of those literals are independent of the environment.
- The clause that evaluates identifiers looks up the identifier in the environment. If ρ_1 is more defined than ρ_0, then the value of `x` in ρ_1 must be at least as well defined as the value of `x` in ρ_0. Therefore, $\mathcal{SE}[\![\ x\ ]\!]\,\rho_0 \ \sqsubseteq \mathcal{SE}[\![\ x\ ]\!]\,\rho_1$.

Now $\mathcal{E}$, by structural induction:

- The clauses that evaluate constants and literals are all monotonic because the values they return are from calls to the simple evaluator, which is monotonic, and the stores they return are the incoming stores.
- The clauses that evaluate arithmetic and relational operators are all monotonic because these operators are monotonic (*e.g.*, $(\perp + 2) \sqsubseteq (3 + 2)$) and the values passed to these operators are obtained from the simple expression evaluator, which is monotonic.
- In the clause that evaluates function applications, the argument values are obtained from the simple expression evaluator, so they increase monotonically as the environment gets more defined. We use our induction hypothesis to show that evaluation of the function body is monotonic, because recursive calls to $\mathcal{E}$ are assumed to be monotonic.
- If we assume that the semantic conditional returns $\perp$ if the predicate is undefined, then as the predicate gets more defined the result of the conditional gets more defined. If the predicate is either `True` or `False`, then the behavior is monotonic because we assumed that the subsequent calls to $\mathcal{E}$ are monotonic.

```
{ def f(x,y) =
    { t = l0MakeTuple(x,y)
      result = k0g(t);
     ---
      Dealloc(t);
    in result };

  def g(t) =
    Select1(t)
}
```

Figure 2.4: Simple deallocation example

- To show that evaluation of `letrec` blocks is monotonic, we must show that function *EvalBindings* is monotonic. Because $\mathcal{E}$ is extensive, each of the new stores created in *EvalBindings* is at least as defined as the incoming store, so *EvalBindings* is monotonic in the return store. *EvalBindings* is monotonic in the return environment because the new environment is created by binding each variable x_i to the least upper bound of the new approximation of its value and its binding in the previous environment. Evaluation of `letrec` blocks is monotonic because *EvalBindings* is monotonic and because we do not remove the binding of a label from the store when it is deallocated.
- Evaluation of each of the allocation primitives is monotonic because they are extensive with respect to the stores, and because they use the simple expression evaluator to evaluate their arguments.
- Evaluation of each of the selection primitives is monotonic because they return the value of a structure from the input store. If the store becomes more defined then the value of a structure in the store must stay at least as well defined as it was before.

■

2.5 The Deallocation Problem

We are trying to solve two related problems. One problem is: given a program with deallocation statements in it, verify the correctness of those deallocation statements. The second problem is to insert deallocation statements into a program automatically.

In either case, we must know when a deallocation command is correct. In the program in Figure 2.4, procedure `f` contains a statement deallocating the object bound to variable `t`. This statement is correct only if the structure to which `t` is bound was allocated within the body of `f`, the structure does not escape from the result of `f`, and there is no other statement deallocating that structure.

Thus, we are interested in four important bits of information for any program or procedure in a program:

- the identities of the objects to which variables are bound;
- the identities of the objects that procedures allocate;

- the identities of the objects that procedures return; and
- the identities of the objects that procedures deallocate.

The first bit of information is used to determine the other three and to associate lifetime information with the program once it has been analyzed. The second and third bits let us determine which objects are reachable, and potentially live, outside of the current procedure activation. These two pieces of information are also used to determine which objects have lifetimes completely bounded by the lifetime of a procedure's activation frame. Given more precise information about the order of execution of a procedure body, its arguments, and its child procedure calls, we could perform better dependence analysis that would tell us which objects that are live in this procedure activation frame are needed after termination of this activation frame. The last bit of information is necessary in order to prevent errors that can occur if the heap manager is requested to deallocate the same object more than once.

2.6 Overview of Our Solution

The goal of this thesis is to develop an analysis that yields the necessary information to verify or insert storage reclamation code. In the next three chapters, we develop a solution to the problem of determining object lifetimes at compile-time. Chapter 3 describes an interpreter for KID$^-$ that allows us to determine the unique identities of objects at run-time and to determine exactly when these objects are allocated and when they are no longer reachable. Chapter 4 describes an abstraction of this semantics that allows us to compute a generalization of the lifetimes of objects over all executions of a program. Chapter 5 gives algorithms for verifying and inserting deallocation statements using information from lifetime analysis. Chapters 6, 7, 8, and 9 extend the value domains and the standard, instrumented, and abstract interpreters to handle arrays, lists, and I-structures. Chapter 10 will describe the compile and run-time performance of programs automatically annotated by the compiler and Chapter 11 presents the conclusions we have reached during this work.

Chapter 3

Instrumented Semantics

Before we can formalize the conditions that must be satisfied statically by a correct storage deallocation command, we must know the conditions that must be satisfied dynamically for that deallocation command to be correct. The standard KID$^-$ interpreter treats the deallocation primitive as a no-op, and so it is not sufficient for our purposes.

In order to determine that a deallocation command is correct, we must be able to determine that no reference is made to an object after it is deallocated. In a sequential interpreter, we could mark the location when it was deallocated, and any further reference to that location would produce an error. In our interpreter, however, we cannot mark an object as deallocated because of the way we evaluate `letrec` blocks — stores are repeatedly passed through all subexpressions of the block.

In this chapter, the standard semantics will be augmented to collect information about which objects were allocated, dereferenced or deallocated by each expression. These collections of events can be examined after a program has been interpreted to see if any object was dereferenced after it was deallocated.

The activation labels defined in Chapter 2 give a partial order on the time of execution of the instances of each subexpression in a program. In this chapter, we will assign new activation labels for the body of each `letrec` block as well as for each procedure application, so that we can measure finer differences in execution times. Activation labels as defined earlier are sufficient to distinguish each object that is created by a program, but they are not sufficient to distinguish in which control region a particular deallocation takes place.

In the first section of this chapter, we will see how the information we would like to gather affects the structure of the instrumented interpreter. In the next section, we will present the instrumented interpreter. Following that, we will discuss the correctness of the interpreter with respect to the standard interpreter given in Chapter 2. Finally, we will work through the interpretation of a few examples.

3.1 Instrumented Interpreter Characteristics

The four pieces of information needed to verify the correctness of the deallocation of the structure bound to a variable during the execution of a program will help us define the domains of the instrumented interpreter and the signatures of its functions.

3.1.1 Collecting the Necessary Information

First, we must be able to identify individual objects in a program in order to determine when two variables are bound to the same object. We use the object labels defined in Chapter 2 to name objects uniquely. Although the activation label component of the object label has structure that allows us to determine the relative timing of object allocation and deallocation, we only consider equality of labels, not ordering on the structure of labels, when we manipulate sets of object labels. For instance, when we take the union of two sets of labels, we return a set containing all of the different labels. We do use the structure of individual labels as our notion of time of execution, though.

Second, we must collect the labels of objects allocated during the evaluation of an expression. In the standard interpreter for KID$^-$, each expression evaluates to a single complete value: a denotable value and a store. In our instrumented interpreter, each expression evaluates to a denotable value, a store, and three sets of events. These event collections name the objects that were allocated, deallocated, and referenced, and the activations in which the event occurred.

Finally, we can examine the values to which expressions evaluate in order to see what locations may be reachable from the result of an expression. The result of interpretation of an expression is a denotable value and a store. We can traverse the value with respect to the store to determine the set of reachable locations. This information will be used to formulate a conservative safety condition for deallocation statements. We will be able to test this condition in the context of the deallocation statement, rather than during a postmortem after execution occurs (as is required when examining the allocation, deallocation, and reference events).

3.1.2 Temporal Ordering of Execution

Once we have collected sets of allocation, deallocation, and dereferencing events, the next step is to give a partial order on the execution of these events. Activation labels have structure that allows us to use the hierarchical termination of activations as a measure of execution time. We use that to order events in time.

In the instrumented interpreter, every distinct activation label names a different control region. We extend the invocation and termination precedence relations from control regions to activation labels. Thus, we say that an activation labeled α_0 *terminates before* an activation labeled α_1 if the termination of the control region labeled α_0 must precede the termination of the control region labeled α_1. In other words, α_0 is a prefix of α_1, that is, the control region, or activation, named by α_0 is an ancestor of the activation named by α_1. In the KID$^-$ interpreter, termination proceeds hierarchically — parent activations cannot terminate until all of their child activations have terminated.

Every `letrec` block in KID$^-$ has a group of bindings, a barrier, and a group of deallocation commands. If the block expression has label k and is executing in activation α, then $\alpha.k$ will be the label of the control region containing the group of bindings, and $\alpha.k^-$ will be the label of the group of deallocation commands. These two activation label satisfy the relation: $(\alpha.k \texttt{---} \alpha.k^-)$.

Definition 3.1 (Activation Label Termination Order) *The relation $\alpha_0 \preceq_T \alpha_1$ means that activation α_0 must terminate before activation α_1 and is defined as follows:*

$$\alpha_0 \preceq_T \alpha_1 \equiv (\alpha_0 = \alpha_1.\beta)$$

where β is a string of zero or more expression labels.

$n \in N$	$=$	$\{\cdots, -1, 0, 1, \cdots\}$	Integers
$b \in B$	$=$	$\mathsf{True} + \mathsf{False}$	Booleans
$l \in L$	$=$	$\{l_0, l_1, l_2, \cdots\}$	Expression labels
$\alpha \in AL$	$=$	$\epsilon\ +\ AL.L$	Activation Labels
$ol \in OL$	$=$	$AL : L$	Object labels
$v \in V$	$=$	$(N + B + OL)_\perp$	Denotable values
$t \in Tuple$	$=$	$\langle_{Tuple}\ V, \cdots, V\rangle$	Tuples
$v_{array} \in Array$	$=$	$\langle_{Array}\ N, V, \cdots, V\rangle$	Arrays
$v_{oneof} \in Oneof$	$=$	$\langle_{N,N}\ V, \cdots, V\rangle$	Oneofs
$v_{list} \in List$	$=$	$\langle_{Cons}\ V, OL\rangle + \langle_{Nil}\ \rangle$	Lists
$sv \in SV$	$=$	$(Tuple + Array + Oneof + List)_\perp$	Storable Values
$\sigma \in Store$	$=$	$OL \rightarrow SV$	Stores
$\rho \in Env$	$=$	$X \rightarrow V$	Environments

Figure 3.1: Instrumented semantic domains

In other words, activation α_1 must terminate before activation α_0 terminates if α_0 is an ancestor of α_1. If activation α_0 is preceded by α_1, then we will say that α_0 is an ancestor of α_1, or that α_0 is higher in the call tree than α_1.

We will use this notion of termination order to catch dangling pointer errors, and to give correctness conditions on programs to guarantee that no such errors will occur at run-time.

3.2 An Instrumented Interpreter

Now that we have formulated some of the criteria that the instrumented interpreter must satisfy, let us develop the interpreter and its value domains in more detail. In this section we define an instrumented interpreter for KID$^-$ based on the ideas presented earlier in this chapter and in the previous chapter.

3.2.1 Semantic Domains

As in the standard semantics, the instrumented semantics operates over integers, booleans, tuples, arrays, and lists. We will use the domains from Chapter 2, which are shown again for reference in Figure 3.1. The domain ordering and least upper bound were shown in Figure 2.2.

In addition to the values and stores computed by the standard interpreter, the instrumented interpreter three sets of *events*. An object event pairs the object label of an object that was allocated, deallocated, or dereferenced, and the activation label in which the allocation, deallocation, or dereferencing occurred.

The domains of allocation events ($AEVs$), deallocation events ($DEVs$), and dereferencing events ($REVs$) are defined as follows:

$$\begin{aligned} AEVs &= \mathcal{P}(OL \times AL) \\ DEVs &= \mathcal{P}(OL \times AL) \\ REVs &= \mathcal{P}(OL \times AL) \end{aligned}$$

Each type of event consists of an object label paired with the activation label denoting when that event occurred. We will refer to allocation, deallocation, and dereferencing events collectively as *object events*. The interpreter will collect sets of events, rather than sequences, because not all events can be ordered.

3.2.2 Semantic Functions

This section presents the definition of the instrumented interpreter, which augments the standard interpreter with mechanisms to collect object events.

The following are the semantic functions that make up the instrumented interpreter:

$$\begin{array}{l} \mathcal{E}_I : E \rightarrow Env \rightarrow Store \rightarrow AL \rightarrow (V \times Store \times AEVs \times DEVs \times REVs) \\ \mathcal{PE}_I : Prog \rightarrow (V \times Store \times AEVs \times DEVs \times REVs) \end{array}$$

The three extra values returned by the interpreter: $\Delta^+ \in AEVs$, $\Delta^- \in DEVs$, and $\Delta^R \in REVs$, tell us exactly which objects were allocated, deallocated, and dereferenced in each instance of each expression.

The function $\mathcal{E}_I$ takes an expression, an environment, a store and an activation label, and returns the resulting value, the resulting store and the sets of allocation, deallocation, and dereferencing events yielded by the interpretation of that expression. The function $\mathcal{PE}_I$ takes a complete program and returns the result value and store and the set of allocation, deallocation, and dereferencing events from the execution of the program.

Note that $\mathcal{E}_I$, like $\mathcal{E}$, is extensive with respect to stores and also monotonic. These properties are necessary in order to prove that the instrumented interpreter terminates with a unique result.

Program Evaluator Definition

The definition of the program evaluator is almost exactly like that of the standard program evaluator, except that it returns three sets of object events. Here is the definition of $\mathcal{PE}_I$, which interprets programs.

$$\begin{array}{l} \mathcal{PE}_I [\![\ \{ \cdots f_i(x_{i,1}, \ldots, x_{i,n_i}) = e_i \ \cdots \} \ ]\!] = \\ \quad \{ \ \langle \ v, \sigma, \Delta^+, \Delta^-, \Delta^R \rangle = \mathcal{E}_I [\![\ e_0 \ ]\!] \perp_{Env} \perp_{Store} \epsilon \, ; \\ \quad\ \ \text{in} \ \langle \ v, \sigma, \Delta^+, \Delta^-, \Delta^R \rangle \ \} \end{array}$$

where expression e_0 is the body of the main procedure f_0.

Simple Expression Evaluator Definition

The instrumented interpreter uses the simple expression evaluator from the standard interpreter. This is shown for reference in Figure 3.2.

$$
\begin{array}{rcll}
\mathcal{SE}[\![\ n\ ]\!]\rho & = & \underline{n} & \text{where } n \text{ is a number} \\
\mathcal{SE}[\![\ b\ ]\!]\rho & = & \underline{b} & \text{where } b \text{ is a boolean} \\
\mathcal{SE}[\![\ x\ ]\!]\rho & = & \rho[x] & \text{where } x \text{ is a variable}
\end{array}
$$

Figure 3.2: Simple expression evaluator

$$
\begin{array}{rcl}
\mathcal{E}_I[\![\ n\ ]\!]\rho\sigma\alpha & = & \langle\ \mathcal{SE}[\![\ n\ ]\!]\rho\ , \sigma, \emptyset, \emptyset, \emptyset\ \rangle \\
\mathcal{E}_I[\![\ b\ ]\!]\rho\sigma\alpha & = & \langle\ \mathcal{SE}[\![\ b\ ]\!]\rho\ , \sigma, \emptyset, \emptyset, \emptyset\ \rangle \\
\mathcal{E}_I[\![\ x\ ]\!]\rho\sigma\alpha & = & \langle\ \mathcal{SE}[\![\ x\ ]\!]\rho\ , \sigma, \emptyset, \emptyset, \emptyset\ \rangle
\end{array}
$$

$$
\mathcal{E}_I[\![\ +(se_1, se_2)\ ]\!]\rho\sigma\alpha \quad = \quad \langle\ \mathcal{SE}[\![\ se_1\ ]\!]\rho\ + \mathcal{SE}[\![\ se_2\ ]\!]\rho\ , \sigma, \emptyset, \emptyset, \emptyset\ \rangle
$$

Figure 3.3: Evaluation of simple expressions and primitive operators

Expression Evaluator Definition

In this section we discuss the definition of the instrumented expression evaluator.

Simple expressions and primitive arithmetic and boolean operators are evaluated in a manner similar to that of the standard interpreter. The result is a quintuple consisting of the value, the incoming store, and three empty sets because simple expressions cannot update the store, or allocate, deallocate or dereference locations. These four clauses of the interpreter are shown in Figure 3.3.

The clauses for evaluation of function applications and conditionals are shown in Figure 3.4. These clauses are the same as the corresponding clauses from the standard interpreter, except that evaluation of the body of the function and the taken branch of a conditional yield sets of object events.

Evaluation of `letrec` blocks in the instrumented interpreter is similar to evaluation of `letrec` blocks in the standard interpreter, except that this interpreter must collect the sets of labels of objects allocated and deallocated by each binding right-hand-side. In addition, the body of the `letrec` block in the instrumented semantics will be evaluated in a new activation, whose label is the `letrec` block's expression label concatenated to the current activation label. This new activation label gives us a more precise notion of when objects are allocated, deallocated, and dereferenced. This information will be used to determine if any dangling pointer errors occur. The labels of objects deallocated by the deallocation statements of a `letrec` block are returned with the set of labels of objects deallocated during execution of the bindings. The interpreter clause for `letrec` expressions is shown in Figure 3.5.

Figure 3.6 gives the evaluation rules for tuple data structures. These are similar to the corresponding clauses of the standard interpreter, except that they return object events. MakeTuple returns the same value and store in the instrumented interpreter as in the standard interpreter, but it also returns three sets of object events. The dereferencing and deallocation event sets are both empty, but the allocation event set consists of a single element: the object label paired with the current activation label. The primitive $\texttt{Select}_i$ returns the ith component of the tuple, the incoming store, empty sets of allocation and deallocation events, and a dereferencing event set consisting of a single element: the object label of the argument paired with the current activation label.

$$
\begin{aligned}
&\mathcal{E}_I [\![\ {}^k f(se_1, \cdots, se_n) \ ]\!] \rho\sigma\alpha \ = \\
&\qquad \{ \ v_1 \qquad\qquad\qquad\quad = \mathcal{SE} [\![\ se_1 \ ]\!] \rho \ ; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\quad v_n \qquad\qquad\qquad\quad = \mathcal{SE} [\![\ se_n \ ]\!] \rho \ ; \\
&\qquad\quad \alpha' \qquad\qquad\qquad\quad = \alpha.k; \\
&\qquad\quad \langle \ v, \sigma', \Delta^+, \Delta^-, \Delta^R \rangle = \mathcal{E}_I [\![\ e \ ]\!] (\rho[v_1/x_1, \cdots, v_n/x_n])\sigma\alpha' \ ; \\
&\qquad\quad \text{in } \langle \ v, \sigma', \Delta^+, \Delta^-, \Delta^R \rangle \ \} \\
&\qquad \text{where } f(x_1, \cdots, x_n) = e \text{ is a definition in the program} \\
&\mathcal{E}_I [\![\ \texttt{if}\,(se_0, e_1, e_2) \ ]\!] \rho\sigma\alpha \ = \\
&\qquad \textit{if } \mathcal{SE} [\![\ se_0 \ ]\!] \rho \\
&\qquad\quad \textit{then } \mathcal{E}_I [\![\ e_1 \ ]\!] \rho\sigma\alpha \\
&\qquad\quad \textit{else } \mathcal{E}_I [\![\ e_2 \ ]\!] \rho\sigma\alpha
\end{aligned}
$$

Figure 3.4: Evaluation of conditional expressions

Figure 3.7 contains the clauses of $\mathcal{E}_I$ for the array primitives. These clauses are the same as the clauses from the standard interpreter except that they return sets of allocation, deallocation, and dereferencing events in addition to a value and store. The primitive $\texttt{MakeArray}_{f_i}$ collects the allocation, deallocation, and dereferencing events from each of the calls to f_i and augments the set Δ^+ of allocation events to include the allocation of object *ol* in activation α. The `Fetch` and Bounds primitives record that the label *ol* of the array passed to them was referenced in the current activation α in Δ^R, the set of reference events that they return.

The evaluator clauses for algebraic types, given in Figure 3.8, and the evaluator clauses for list primitives, given in Figure 3.9, are similar to the corresponding clauses from the standard interpreter, except that the constructors return non-empty allocation event sets, and the selectors and predicates return non-empty dereferencing event sets.

3.2.3 Correctness of the Interpreter

We will consider the instrumented interpreter to be correct if the denotable value and the store returned from the execution of a program under the instrumented interpreter are always equal to the denotable value and store returned by the execution of the program under the standard interpreter.

Theorem 3.2 *The instrumented interpreter is correct with respect to the standard interpreter.*

$$
\begin{aligned}
&\forall \ pr \in \mathit{Prog}, \\
&\exists \langle \ v_s, \sigma_s \ \rangle \ = \mathcal{PE} [\![\ pr \ ]\!] \ , \\
&\exists \langle \ v_t, \sigma_t, \Delta^+, \Delta^-, \Delta^R \rangle \ = \mathcal{PE}_I [\![\ pr \ ]\!] \ , \\
&(v_s = v_t) \wedge (\sigma_s = \sigma_t)
\end{aligned}
$$

Proof:

Informally: The instrumented interpreter computes the same values and stores as the standard interpreter, because all portions of the interpreter that compute values and stores are the same as the standard interpreter. ■

$$
\begin{array}{l}
\mathcal{E}_I[\![\ {}^k\{\ Bs\text{—}Ds\ \mathtt{in}\ x\}\ ]\!]\rho\sigma\alpha\ = \\
\quad \{\ [\![\ x_1 = e_1; \ldots; x_n = e_n\ ]\!] = Bs; \\
\quad\ \ [\![\ \mathtt{Dealloc}(y_1); \cdots; \mathtt{Dealloc}(y_k)\ ]\!] = Ds; \\
\quad\ \ \begin{array}{ll}
\rho_0 & = \rho[\bot/x_1, \cdots, \bot/x_n]; \\
\alpha' & = \alpha.k; \\
\langle\ \rho', \sigma', \Delta^{+\prime}, \Delta^{-\prime}, \Delta^{R\prime}\ \rangle & = \mathit{EvalBindings}_I(Bs, \rho_0, \sigma, \alpha'); \\
\Delta^{-\prime\prime} & = \Delta^{-\prime} \cup \{\langle\ \rho'[y_i],\ \alpha'\ \rangle\ |1 \le i \le k\}\,; \\
\end{array} \\
\quad\ \ \text{in}\ \langle\ \rho'[x], \sigma', \Delta^{+\prime}, \Delta^{-\prime\prime}, \Delta^{R\prime}\ \rangle\ \}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{EvalBindings}_I([\![\ x_1 = e_1; \ldots; x_n = e_n\ ]\!], \rho, \sigma, \alpha) = \\
\quad \{\ \langle\ v_1, \sigma_1, \Delta^+{}_1, \Delta^-{}_1, \Delta^R{}_1\ \rangle\ = \mathcal{E}_I[\![\ e_1\ ]\!]\rho\sigma\alpha\,; \\
\qquad\qquad \vdots \\
\quad\ \ \langle\ v_n, \sigma_n, \Delta^+{}_n, \Delta^-{}_n, \Delta^R{}_n\ \rangle\ = \mathcal{E}_I[\![\ e_n\ ]\!]\rho\sigma\alpha\,; \\
\quad\ \ \rho'\ \ \ = \rho[(v_1 \sqcup \rho[x_1])/x_1, \cdots, (v_n \sqcup \rho[x_n])/x_n]; \\
\quad\ \ \sigma'\ \ \ = \bigsqcup_i \sigma_i; \\
\quad\ \ \Delta^{+\prime} = \bigcup_i \Delta^+{}_i; \\
\quad\ \ \Delta^{-\prime} = \bigcup_i \Delta^-{}_i; \\
\quad\ \ \Delta^{R\prime} = \bigcup_i \Delta^R{}_i; \\
\quad\ \ \langle\ \rho'', \sigma'', \Delta^{+\prime\prime}, \Delta^{-\prime\prime}, \Delta^{R\prime\prime}\ \rangle\ = \\
\qquad\qquad \mathit{if}\ \rho' = \rho \wedge \sigma' = \sigma \\
\qquad\qquad\ \ \mathit{then}\ \langle\ \rho', \sigma', \Delta^{+\prime}, \Delta^{-\prime}, \Delta^{R\prime}\ \rangle \\
\qquad\qquad\ \ \mathit{else}\ \mathit{EvalBindings}_I([\![\ x_1 = e_1; \ldots; x_n = e_n\ ]\!], \rho', \sigma', \alpha) \\
\quad\ \ \text{in}\ \langle\ \rho'', \sigma'', \Delta^{+\prime\prime}, \Delta^{-\prime\prime}, \Delta^{R\prime\prime}\ \rangle\ \}
\end{array}
$$

Figure 3.5: Evaluation of block expressions

3.2.4 Soundness of the Instrumented Interpreter

Theorem 3.3 *The instrumented interpreter $\mathcal{E}_I$ is extensive with respect to stores.*

$$
\begin{array}{l}
\forall e \in E,\ \forall \alpha inAL,\ \forall \rho \in \mathit{Env},\ \forall \sigma_0 \in \mathit{Store}, \\
\exists\,\langle\ v, \sigma_1, \Delta^+, \Delta^-, \Delta^R\ \rangle\ = \mathcal{E}_I[\![\ e\ ]\!]\rho\sigma_0\alpha\,, \\
\ \sigma_0 \sqsubseteq \sigma_1
\end{array}
$$

Proof:

Similar to the proof of the extensionality of the standard interpreter. ■

Theorem 3.4 *Interpreter function $\mathcal{E}_I$ is monotonic with respect to the context:*

$$
\begin{array}{l}
\forall e \in E,\ \forall \alpha \in AL,\ \forall \rho_0, \rho_1 \in \mathit{Env},\ \forall \sigma_0, \sigma_1 \in \mathit{Store}, \\
\rho_0 \sqsubseteq \rho_1 \wedge \sigma_0 \sqsubseteq \sigma_1 \Rightarrow \mathcal{E}_I[\![\ e\ ]\!]\rho_0\sigma_0\alpha\ \sqsubseteq \mathcal{E}_I[\![\ e\ ]\!]\rho_1\sigma_1\alpha
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_I [\![\; {}^{l}\texttt{MakeTuple}(se_1, \cdots, se_m) \;]\!] \rho\sigma\alpha \; = \\
\qquad \{ \; v_1 \quad = \mathcal{SE} [\![\; se_1 \;]\!] \rho \; ; \\
\qquad\qquad\qquad \vdots \\
\qquad\quad v_n \quad = \mathcal{SE} [\![\; se_n \;]\!] \rho \; ; \\
\qquad\quad ol \quad = \alpha : l; \\
\qquad\quad v_{tuple} = \langle_{Tuple} \; v_1, \cdots, v_m \rangle \; ; \\
\qquad\quad \sigma' \quad = \sigma[ol \to v_{tuple}]; \\
\qquad\quad \text{in} \; \langle \; ol, \sigma', \{ \langle \; ol, \; \alpha \; \rangle \}, \emptyset, \emptyset \; \rangle \; \} \\
\mathcal{E}_I [\![\; \texttt{Select}_i(se) \;]\!] \rho\sigma\alpha \; = \\
\qquad \{ \; ol \qquad\qquad\qquad = \mathcal{SE} [\![\; se \;]\!] \rho \; ; \\
\qquad\quad \langle_{Tuple} \; v_1, \cdots, v_m \rangle = \sigma[ol]; \\
\qquad\quad \text{in} \; \langle \; v_i, \sigma, \emptyset, \emptyset, \{ \langle \; ol, \; \alpha \; \rangle \} \; \rangle \; \}
\end{array}
$$

Figure 3.6: Evaluation of tuple primitives

Proof:

Similar to the proof of the monotonicity of the standard interpreter. ■

3.3 Interpretation of Some Examples

In this section we will evaluate a couple of examples under the instrumented interpreter to illustrate its behavior.

3.3.1 Interpretation of a Non-Recursive Example

We will start with a non-recursive example:

```
{ def f(w) =
    k2{ t = k0g(w);
       a = Select1(t);
       b = Select2(t);
       r = (a * b);
      in r };
  def g(x) =
    k3{ y = (x-21);
        t = l0MakeTuple(x,y) ;
     in t }
  def f0 () =
    k1f(68);
};
```

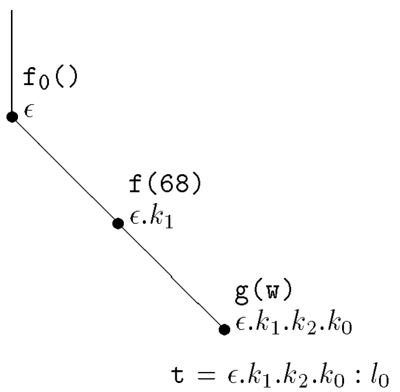
If we execute the program using the instrumented interpreter, we get the following call tree:

$$
\begin{array}{l}
\mathcal{E}_I [\![\ ^k\mathtt{MakeArray}_{f_i}\,(se_0, se_1, \cdots, se_r) \ ]\!] \rho\sigma\alpha \ = \\
\quad \{ \begin{array}[t]{ll}
ol & = \alpha : k; \\
n & = \mathcal{SE} [\![\ se_0 \ ]\!] \rho \ ; \\
v_1 & = \mathcal{SE} [\![\ se_1 \ ]\!] \rho \ ; \\
 & \ \vdots \\
v_r & = \mathcal{SE} [\![\ se_r \ ]\!] \rho \ ; \\
\alpha_0 & = \alpha.k.(0); \\
\multicolumn{2}{l}{\langle \ u_0, \sigma_0, \Delta^+{}_0, \Delta^-{}_0, \Delta^R{}_0 \ \rangle \ =} \\
 & \qquad \mathcal{E}_I [\![\ e_i \ ]\!] \perp_{Env}[\underline{0}/x_0, v_1/x_1, \cdots, v_r/x_r]\sigma\alpha_0 \ ; \\
 & \ \vdots \\
\alpha_{n-1} & = \alpha.k.(n-1); \\
\multicolumn{2}{l}{\langle \ u_{n-1}, \sigma_{n-1}, \Delta^+{}_{n-1}, \Delta^-{}_{n-1}, \Delta^R{}_{n-1} \ \rangle \ =} \\
 & \qquad \mathcal{E}_I [\![\ e_i \ ]\!] \perp_{Env}[\underline{(n-1)}/x_0, v_1/x_1, \cdots, v_r/x_r]\sigma\alpha_{n-1} \ ; \\
\sigma' & = \sigma[ol \rightarrow \langle_{Array} \ n, u_0, \cdots, u_{n-1}\rangle] \sqcup \left(\bigsqcup_i \sigma_i \right) ; \\
\Delta^+ & = \{ \langle \ ol, \ \alpha \ \rangle \} \cup (\bigcup_i \Delta^+{}_i) \ ; \\
\Delta^- & = \bigcup_i \Delta^-{}_i; \\
\Delta^R & = \bigcup_i \Delta^R{}_i; \\
\multicolumn{2}{l}{\text{in } \langle \ ol, \sigma', \Delta^+, \Delta^-, \Delta^R \ \rangle \ \}}
\end{array} \\
\quad \text{where } f_i(x_0, x_1, \cdots, x_r) = e_i \text{ is a definition in the program}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_I [\![\ ^k\mathtt{Fetch}\,(se_1, se_2) \ ]\!] \rho\sigma\alpha \ = \\
\quad \{ \begin{array}[t]{l}
ol = \mathcal{SE} [\![\ se_1 \ ]\!] \rho \ ; \\
i \ \ = \mathcal{SE} [\![\ se_2 \ ]\!] \rho \ ; \\
\langle_{Array} \ n, v_0, \cdots, v_{n-1}\rangle = \sigma[ol]; \\
\text{in } \langle \ v_i, \sigma, \emptyset, \emptyset, \{ \langle \ \alpha, \ ol \ \rangle \} \ \rangle \ \}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_I [\![\ ^k\mathtt{Bounds}\,(se) \ ]\!] \rho\sigma\alpha \ = \\
\quad \{ \begin{array}[t]{l}
ol = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
\langle_{Array} \ n, v_0, \cdots, v_{n-1}\rangle = \sigma[ol]; \\
\text{in } \langle \ n, \sigma, \emptyset, \emptyset, \{ \langle \ \alpha, \ ol \ \rangle \} \ \rangle \ \}
\end{array}
\end{array}
$$

Figure 3.7: Instrumented evaluation of array primitives



$$
\begin{array}{l}
\mathcal{E}_I [\![\ {}^{l}\texttt{MakeOneof}_{tag,n_{tags}}(se_1, \cdots, se_m) \ ]\!] \rho\sigma\alpha \ = \\
\quad \{ \ v_1 \quad = \mathcal{SE} [\![\ se_1 \ ]\!] \rho \ ; \\
\qquad\qquad \vdots \ ; \\
\quad\ \ v_{n_i} \quad = \mathcal{SE} [\![\ se_{n_i} \ ]\!] \rho \ ; \\
\quad\ \ ol \quad\ = \alpha : l; \\
\quad\ \ v_{oneof} = \left\langle {}_{tag,n_{tags}} \ v_1, \cdots, v_m \right\rangle ; \\
\quad\ \ \sigma' \quad\ = \sigma[ol \rightarrow (v_{oneof} \sqcup \sigma[ol])]; \\
\quad\ \ \text{in} \ \langle \ ol, \sigma', \{\langle \ ol, \ \alpha \ \rangle\}, \emptyset, \emptyset \ \rangle \ \} \\
\mathcal{E}_I [\![\ \texttt{Is}_{tag}?(se) \ ]\!] \rho\sigma\alpha \ = \\
\quad \{ \ ol = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
\quad\ \ \left\langle {}_{tag',n_{tags}} \ v_0, \cdots, v_m \right\rangle = \sigma[ol]; \\
\quad\ \ b \ = \ \ \mathit{if} \ tag = tag' \\
\qquad\qquad \mathit{then} \ \mathsf{True} \\
\qquad\qquad \mathit{else} \ \mathsf{False}; \\
\quad\ \ \text{in} \ \langle \ b, \sigma, \emptyset, \{\langle \ ol, \ \alpha \ \rangle\}, \emptyset \ \rangle \ \} \\
\mathcal{E}_I [\![\ \texttt{Select}_{tag,i}(se) \ ]\!] \rho\sigma\alpha \ = \\
\quad \{ \ ol = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
\quad\ \ \left\langle {}_{tag',n_{tags}} \ v_1, \cdots, v_m \right\rangle = \sigma[ol]; \\
\quad\ \ v = \ \ \mathit{if} \ tag = tag' \ ; \\
\qquad\qquad \mathit{then} \ v_i \\
\qquad\qquad \mathit{else} \ \bot \\
\quad\ \ \text{in} \ \langle \ v, \sigma, \emptyset, \{\langle \ ol, \ \alpha \ \rangle\}, \emptyset \ \rangle \ \}
\end{array}
$$

Figure 3.8: Instrumented evaluation of oneof primitives

Each node in the call tree is labeled with the expression that invoked the procedure corresponding to that node and with the activation label of that node. We also show the binding of variable `t` in procedure `f` to the tuple allocated within `g`, labeled $\epsilon.k_1.k_2.k_0.k_3 : l_0$.

The result under the instrumented semantics is:

$$
\begin{array}{ll}
\langle & \underline{3196}, \\
& \bot_{Store}[\epsilon.k_1.k_2.k_0.k_3 : l_0 \rightarrow \langle_{Tuple} \ 68, 47\rangle], \\
& \{\langle \ \epsilon.k_1.k_2.k_0.k_3 : l_0, \ \epsilon.k_1.k_2.k_0.k_3 \ \rangle\}, \\
& \emptyset, \\
& \{\langle \ \epsilon.k_1.k_2.k_0.k_3 : l_0, \ \epsilon.k_1.k_2 \ \rangle\} \ \rangle
\end{array}
$$

The first component of the result indicates that the answer was the number 3196. The second component of the result, the store, indicates the store at the end of the evaluation of the example. The third component indicates that a single location, $\epsilon.k_1.k_2.k_0.k_3 : l_0$, was allocated, the fourth component indicates that no locations were deallocated, and the final component indicates that location $\epsilon.k_1.k_2.k_0.k_3 : l_0$ was dereferenced during execution in activation $\epsilon.k_1.k_2$.

In this program, the lifetime of the tuple that `g` allocates is from the time that `g` allocated the tuple until procedure `f` terminates, because that is the last time that there is a pointer to the tuple. We

$$
\begin{array}{rcl}
\mathcal{E}_I [\![\ {}^{l}\texttt{Cons}\,(se_1, se_2) \ ]\!] \rho\sigma\alpha & = & \{ \ v_1 \quad = \mathcal{SE} [\![\ se_1 \ ]\!] \rho \ ; \\
& & \quad v_2 \quad = \mathcal{SE} [\![\ se_2 \ ]\!] \rho \ ; \\
& & \quad v_{list} = \langle_{Cons} \ v_1, v_2 \rangle \ ; \\
& & \quad ol \quad = \alpha : l; \\
& & \quad \sigma' \quad = \sigma[ol \rightarrow v_{list}]; \\
& & \quad \text{in} \ \langle \ ol, \sigma', \{\langle \ ol, \ \alpha \ \rangle\}, \emptyset, \emptyset \ \rangle \ \} \\
\mathcal{E}_I [\![\ \texttt{Hd}\,(se) \ ]\!] \rho\sigma\alpha & = & \{ \ ol = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
& & \quad \langle_{Cons} \ v_1, v_2 \rangle = \sigma[ol]; \\
& & \quad \text{in} \ \langle \ v_1, \sigma, \emptyset, \emptyset, \{\langle \ ol, \ \alpha \ \rangle\} \ \rangle \ \} \\
\mathcal{E}_I [\![\ \texttt{Tl}\,(se) \ ]\!] \rho\sigma\alpha & = & \{ \ ol = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
& & \quad \langle_{Cons} \ v_1, v_2 \rangle = \sigma[ol]; \\
& & \quad \text{in} \ \langle \ v_2, \sigma, \emptyset, \emptyset, \{\langle \ ol, \ \alpha \ \rangle\} \ \rangle \ \} \\
\mathcal{E}_I [\![\ {}^{l}\texttt{Nil}\,() \ ]\!] \rho\sigma\alpha & = & \{ \ ol \quad = \alpha : l; \\
& & \quad v_{list} = \langle_{Nil} \ \rangle \ ; \\
& & \quad \sigma' \quad = \sigma[ol \rightarrow v_{list}]; \\
& & \quad \text{in} \ \langle \ ol, \sigma', \{\langle \ ol, \ \alpha \ \rangle\}, \emptyset, \emptyset \ \rangle \ \} \\
\mathcal{E}_I [\![\ \texttt{Nil?}\,(se) \ ]\!] \rho\sigma\alpha & = & \{ \ ol = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
& & \quad b \ = \ \ \mathit{if} \ \sigma[ol].\mathsf{tag} = \texttt{Nil} \\
& & \qquad\qquad \mathit{then} \ \mathsf{True} \\
& & \qquad\qquad \mathit{else} \ \mathsf{False}; \\
& & \quad \text{in} \ \langle \ b, \sigma, \emptyset, \emptyset, \{\langle \ ol, \ \alpha \ \rangle\} \ \rangle \ \}
\end{array}
$$

Figure 3.9: Instrumented evaluation of list primitives

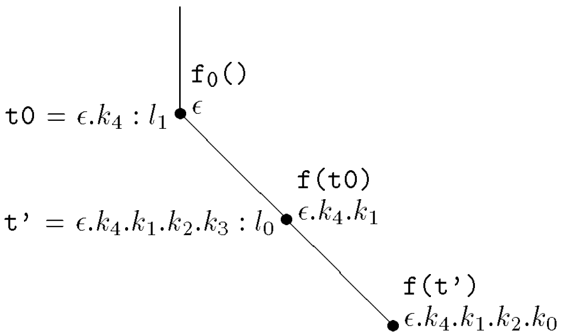
can also say that the lifetime of the tuple labeled $\epsilon.k_1.k_2.k_0.k_3 : l_0$ is bounded by the lifetime of activation $\epsilon.k_1.k_2$.

3.3.2 Interpretation of a Recursive Example

Now let us evaluate a recursive example.

```
{ def foo(t) =
    k2{ a = Select1(t);
        b = Select2(t);
        p = (a == 5);
        r = if p then a
            else k3{ t' = l0MakeTuple(5,7);
                     v = k0foo(t');
                    in v }
      in r }
   def f0 () =
    k4{ t0 = l1MakeTuple(3,4)
        result =  k1foo(t0)
      in result }
}
```

Evaluation of this program under the instrumented interpreter yields the following call tree:



The result under the instrumented semantics is:

$$\begin{array}{ll} \langle & \underline{5}, \\ & \bot_{Store}[\epsilon.k_4.k_1.k_2.k_3 : l_0 \to \langle_{Tuple}\ 4, 5\rangle, \epsilon.k_4 : l_1 \to \langle_{Tuple}\ 3, 4\rangle], \\ & \{\langle\ \epsilon.k_4.k_1.k_2.k_3 : l_0,\ \epsilon.k_4.k_1.k_2.k_3\ \rangle, \langle\ \epsilon.k_4 : l_1,\ \epsilon.k_4\ \rangle\}, \\ & \emptyset, \\ & \{\langle\ \epsilon.k_4 : l_1,\ \epsilon.k_4.k_0.k_2\ \rangle, \langle\ \epsilon.k_4.k_1.k_2.k_3 : l_0,\ \epsilon.k_4.k_1.k_2.k_3.k_0.k_2\ \rangle\}\ \rangle \end{array}$$

which shows that the result was the number 5 and that two tuples, labeled $\epsilon.k_4 : l_1$ and $\epsilon.k_4.k_1.k_2.k_3 : l_0$, were allocated. Neither of these tuples is reachable from the result, and no tuples were deallocated. The two labels were dereferenced; the object labeled $\epsilon.k_4 : l_1$ was dereferenced in activation $\epsilon.k_4.k_1.k_2$ and the object labeled $\epsilon.k_4.k_1.k_2.k_3 : l_0$ was dereferenced in activation $\epsilon.k_4.k_1.k_2.k_3.k_2$.

3.4 Object Deallocation Safety Condition

A number of definitions are needed before we can give a safety condition for object deallocations.

Given a denotable value v and a store σ, we must be able to determine the labels of the objects reachable from value v in store σ. The following definition defines which objects are reachable from a given dynamic value and store.

Definition 3.5 (Object Reachability) *$Reachable\,(v, \sigma)$, the set of labels of objects reachable from value v with respect to store σ, is defined as follows:*

$$\begin{aligned}
Reachable\,(\bot, \sigma) &= \emptyset \\
Reachable\,(\underline{n}, \sigma) &= \emptyset \\
Reachable\,(\underline{b}, \sigma) &= \emptyset \\
Reachable\,(ol, \sigma) &= \{ol\} \cup SVReachable\,(\sigma[ol], \sigma)
\end{aligned}$$

$$\begin{aligned}
SVReachable\,(\bot, \sigma) &= \emptyset \\
SVReachable\,(\langle_{Tuple}\ v_1, \cdots, v_n\rangle, \sigma) &= \bigcup_i Reachable\,(v_i, \sigma) \\
SVReachable\,(\langle_{Array}\ n, v_1, \cdots, v_n\rangle, \sigma) &= \bigcup_i Reachable\,(v_i, \sigma) \\
SVReachable\,(\langle_{tag,n}\ v_1, \cdots, v_m\rangle, \sigma) &= \bigcup_i Reachable\,(v_i, \sigma) \\
SVReachable\,(\langle_{Cons}\ v_1, v_2\rangle, \sigma) &= Reachable\,(v_1, \sigma) \cup Reachable\,(v_2, \sigma) \\
SVReachable\,(\langle_{Nil}\ \rangle, \sigma) &= \emptyset
\end{aligned}$$

We also need to know what objects are reachable from the context surrounding an expression. We will call these objects the *inherited* objects. These are the objects that an expression can use that were allocated outside of the expression.

Definition 3.6 (Inherited Objects) *The function $Inherited\,(e, \rho, \sigma)$ returns the set of labels of objects reachable from $FV\,(e)$ given environment ρ and store σ:*

$$Inherited\,(e, \rho, \sigma) = \bigcup_{w \in FV\,(e)} Reachable\,(\rho[w], \sigma)$$

Remember that if variable w is unbound in environment ρ, then $\rho[w]$ is bottom.

Previously, we defined a dangling reference, or dangling pointer, to be a pointer that was dereferenced after it was deallocated. A pointer will also be considered dangling if the activation in which the object is deallocated may terminate before the activation in which the object is allocated (because an allocation is another form of dereferencing a pointer). To be more precise, the activation in which an object is allocated or dereferenced must always terminate before the activation in which the object is deallocated.

Definition 3.7 (Dangling Reference) *For a program pr, let*

$$\begin{aligned}
\langle\ v, \sigma, \Delta^+, \Delta^-, \Delta^R\rangle &= \mathcal{PE}_I [\![\ pr\ ]\!] \\
R &= Reachable\,(v, \sigma)
\end{aligned}$$

Then $\mathcal{DP}\ [\![\ pr\ ]\!]$, the set of dangling pointers after the execution of program pr, is defined by:

$$\mathcal{DP}\ [\![\ pr\ ]\!] =$$

$$\left\{ \langle\, ol,\, \alpha_- \,\rangle \;\middle|\; \begin{array}{l} \langle\, ol,\, \alpha_- \,\rangle \in \Delta^- \\ \wedge(ol \in R \\ \;\vee(\langle\, ol,\, \alpha_r \,\rangle \in \Delta^R \wedge \neg(\alpha_r \preceq_T \alpha_-)) \\ \;\vee(\langle\, ol,\, \alpha_+ \,\rangle \in \Delta^+ \wedge \neg(\alpha_+ \preceq_T \alpha_-))) \end{array} \right\}$$

The set of dangling pointer events is the set of all pairs of object labels *ol* and activation labels α_- such that: (1) a reference to *ol* is returned as part of the result of the program, (2) there is a reference to *ol* in some activation α_r that does not terminate before activation α_-, or (3) the activation in which *ol* was allocated does not terminate before activation α_-. Either of these conditions counts as a dangling pointer error.

The deallocation of an object *ol* in program *pr* at activation label α_- is considered *correct* if the pair $\langle\, ol,\, \alpha_- \,\rangle$ does not show up in the set of dangling pointer events resulting from the execution of program *pr*.

Condition 3.8 (Deallocation Correctness) *The deallocation of object ol upon termination of activation α_- is correct if the following condition holds:*

$$\langle\, ol,\, \alpha_- \,\rangle \notin \mathcal{DP}\, [\![\ pr\ ]\!]$$

where pr is the program.

Condition 3.8 is exact, in that any deallocation command that does not lead to dangling pointer errors will be considered correct. However, we have to execute the whole program before we can determine if any deallocation command is correct.

The reason we cannot verify Condition 3.8 as we evaluate each `letrec` block is that an object may be deallocated in some `letrec` block, and returned as part of the result of that block. This deallocation is correct as long as no attempt is ever made to dereference the object once it has been deallocated. To be more precise, the `letrec` block corresponds to one control region, or activation, and we can deallocate the structure in this control region as long as the structure is never dereferenced in a control region that is an ancestor of this one.

When we verify that deallocation commands are correct, we are willing to be a little less precise and to only accept deallocation commands that deallocate objects in the highest control region from which the objects are reachable. This property we call *safety* — if a deallocation command is *safe*, then it is guaranteed to be correct, although some correct deallocation commands are unsafe.

There are two reasons to use the deallocation safety condition rather than the deallocation correctness condition when we test deallocation commands. One is that safety is a local property, and so this allows us to verify the safety of a deallocation command in a procedure without considering all of the places in which the procedure might be called. This point is especially important if the algorithm is to be generalized to an environment including separate compilation. The second reason is that the simplest version of our abstract interpreter summarizes all activation labels by the empty activation label, and so we cannot tell the relative ordering of subexpressions.

An object deallocation is *safe*, or guaranteed to be correct, if the deallocation occurs in the highest dynamic context from which the object is reachable.

Condition 3.9 (Object Deallocation Safety) *It is* safe *to deallocate object ol in context* $\langle\, \rho_-,\, \sigma_-,\, \alpha_- \,\rangle$ *where*

$$\begin{aligned} \langle\, v, \sigma', \Delta^+, \Delta^-, \Delta^R \rangle &= \mathcal{E}_I\, [\![\ e\ ]\!]\, \rho_- \sigma_- \alpha_- \\ R &= \mathit{Reachable}\,(v, \sigma') \end{aligned}$$

if the following condition holds:

$$\begin{array}{ll} & \langle\, ol,\, \alpha_-\, \rangle \in \Delta^- \\ \wedge & ol \notin R \\ \wedge & \forall \langle\, ol,\, \alpha_r\, \rangle \in \Delta^R.\, (\alpha_r \preceq_T \alpha_-) \\ \wedge & \forall \langle\, ol,\, \alpha_+\, \rangle \in \Delta^+.\, (\alpha_+ \preceq_T \alpha_-) \end{array}$$

and if there is only one deallocation of ol, which is in activation α_-.

This condition is correct whenever an object is deallocated in the highest control region from which it is reachable. For instance, it is always safe to deallocate an object that is not part of the result of a program upon termination of the main procedure $\mathtt{f}_0$, although this may not be of much use.

Unlike Condition 3.8, Condition 3.9 can be checked at the time the deallocate is performed by examining the current program state: the environments of enclosing contexts and the objects reachable from those contexts and the current block expression. For this reason, this condition will be used in Chapters 4 and 5 to develop a static analysis for verifying and inserting deallocation commands.

Theorem 3.10 (Deallocation Safety Theorem) *If an object deallocation satisfies Condition 3.9 (Deallocation Safety), then it satisfies Condition 3.8 (Deallocation Correctness).*

Proof:

> *Sketch of proof:* If an object *ol* is deallocated in activation α_-, the highest activation from which *ol* is reachable, then the allocation of *ol* and all dereferencing of *ol* must take place in activations labeled α_r such that each α_r terminates before α_-. ■

In the next two chapters, we abstract the instrumented interpreter and restate the safety condition in terms of the abstract interpreter. The next two chapters restrict storable values to include only tuples so that we can concentrate on the process of abstraction and how to state and test the deallocation safety condition. Later in the thesis we add the abstraction of other types of objects to our abstract interpreter.

Chapter 4

Abstracted Semantics

The purpose of abstract interpretation is to capture information about the execution of an expression or program over all possible data. We summarize, or *abstract* the values produced by a program over all executions of a program. For instance, if a program evaluates to a number under the standard or instrumented interpreters, our abstract interpreter summarizes its result as $\underline{N}$, meaning any number. Similarly, our abstract interpreter evaluates both branches of all conditionals in order to summarize the behavior of the conditionals. In this way, the behavior over all control paths and over all data can be approximated.

The abstract semantics that we use captures information about the shape and identities of objects that are allocated and the dynamic reachability of these objects from the variables and structures to which they are bound. In the Chapter 5, we use this information about reachability and object identities to develop algorithms to verify the safety of deallocation commands and to insert deallocation commands in KID$^-$ programs.

In the rest of this chapter, we develop an abstracted interpreter that summarizes the behavior of programs in such a way that we can determine object lifetimes. In the first section, we briefly describe how the abstract interpreter is used. In the second section, we define the abstract value domains for this abstract interpreter. In the third section, we describe the evaluation strategy used by our abstract interpreter. In the fourth section, we define the abstract interpreter itself. In the final section, we show some examples of using this interpreter to determine that lifetimes of particular objects are bounded by the lifetimes of given procedure invocations.

4.1 Using the Abstract Interpreter

Our abstract interpreter does not directly yield lifetime information. It computes the shape of the objects that a program may allocate and how they may be interconnected rather than the actual values that may fill those objects. However, a lifetime analyzer uses the connectivity, or reachability, information to determine the approximate lifetimes of objects. This intuition lead to the development of our abstract interpreter and is also the reason why our abstract interpreter is general enough to be used for other analyses.

To perform lifetime analysis on a procedure, we need to know all the possible values to which each variable may be bound and all possible values each object may contain. The questions we ask to determine object lifetimes are, "When is the first possible time that this object is reachable from the running program?" and, "When is the last possible time that this object is reachable from the running program?" Thus, we must be able to know all possible places from which we can reference an object. In the remainder of this section we discuss the precision of this reachability information and how to use that information to determine that deallocation commands are safe.

4.1.1 Precision of Information

The reachability information generated by the abstract interpreter is approximate. However, the imprecision is asymmetrical — a negative result is definite, while a positive result is indefinite. The most precise fact we can determine is that an object is not reachable from a given variable. If the abstract interpreter determines that an abstract object is not reachable from the result of a procedure invocation, then under no circumstances will that object be reachable during execution of the procedure invocation under the standard interpretater. We must be very careful to base all of our decisions on precise negative information rather than approximate positive information. For example, if we determine that variable x may be bound to some set of abstract objects labeled ls, then x may be bound to $\perp$, or to one of the locations in ls, but x will definitely not be bound to a location outside of ls.

Given this insight into the kinds of questions we may ask about object reachability, let us reexamine the three conditions that an object must satisfy in order to be safely deallocated within a dynamic context. First, the object must have been allocated within the context. In other words, the object must not have been inherited, or passed in from a surrounding context. We verify this by testing that the object cannot be reached from a surrounding context. Since we are talking about the binding of a variable, we must actually test that none of the objects to which the variable could be bound can be reached from a surrounding context. Next, the object must not escape from this context. We verify this by testing that none of the object labels to which this variable could be bound are reachable from the result of the context of interest. Finally, this object must not be deallocated more than once. We test this by verifying that none of the object labels to which this identifier may be bound are in the set of object labels that may be deallocated by other deallocation commands.

4.1.2 The Abstract Deallocation Safety Condition

The canonical form of a block expression is shown below:

```
{ x0 = e0;
    ...
  x(n-1) = e(n-1);
 ---
  Dealloc(y0);
    ...
  Dealloc(y(m-1));
in x }
```

We use x_i for the names of the bound variables, y_i for the variables in deallocation commands, and w_i as the free variables of a `letrec` expression. Here, each variable x_i is bound in the block expression. The object to which each of the y_i's is bound will be deallocated once the bindings have completely evaluated, and the value of x is returned as the result of the block expression.

The context in which an expression is evaluated provides the environment, store, and activation label in which the expression is executed. Let us consider a block expression e evaluated in the standard context $c = \langle\, \rho,\, \sigma,\, \alpha \,\rangle$. If there are deallocation commands for $y_1, \cdots, y_n$ in the top level of the block expression, then we can verify that these deallocation commands are correct under the standard interpreter as follows. First, we determine what locations are passed into e from the context. Call this set I.

$$I \;=\; \bigcup_{w \in FV\,(e)} \mathit{Reachable}\,(\rho[w], \sigma) \tag{4.1}$$

We can use either σ or σ' here because the language is functional. If side effects were added we would have to use σ', although we would still use environment ρ.

We evaluate the bindings of the block expression, yielding the environment and store for the evaluation of the body of the expression — call these ρ' and σ'. The resulting value of this evaluation is $\langle\, \rho'[x],\, \sigma' \,\rangle$, where variable x is the result of the block expression.

Now, we must also determine the set R, which is the set of objects reachable from the result of the evaluation of e in context c.

$$R \;=\; \mathit{Reachable}\,(\rho'[x], \sigma') \tag{4.2}$$

where x is the result of the block expression above.

Given this exact information from the standard evaluator — ρ, σ, I, R, ρ', and σ' — we can determine that it is safe to execute each of the deallocation commands in e in a particular dynamic context.

$$\mathit{Safe?}\,([\![\; \texttt{Dealloc}\,(y_i)\;]\!]) \;=\; \left(\begin{array}{l} \rho'[y_i] \notin I \\ \wedge \rho'[y_i] \notin R \\ \wedge \bigwedge_{y_j \neq y_i} \rho'[y_i] \neq \rho'[y_j] \end{array}\right) \tag{4.3}$$

In other words, each deallocation is guaranteed to be correct if the value of y_i — the object being deallocated — is not inherited from the context (it must have been allocated within e), is not returned as part of the result of e, and is not deallocated by any other deallocation command.

In order to verify deallocation safety in the abstract interpreter, we must perform a similar test. So, starting with an abstract context $\langle\, \widehat{\rho},\, \widehat{\sigma},\, \widehat{\alpha} \,\rangle$, we must evaluate the bindings of e to obtain the environment and store ($\widehat{\rho}'$ and $\widehat{\sigma}'$) of the body of the block expression, and then compute sets I and R:

$$I \;=\; \bigcup_{w \in FV\,(e)} \mathit{Reachable}\,(\widehat{\rho}[w])\widehat{\sigma} \tag{4.4}$$

$$R \;=\; \mathit{Reachable}\,(\widehat{\rho}'[x])\widehat{\sigma}' \tag{4.5}$$

Given all of these abstract values, we can conservatively determine safety using the following procedure:

$$\widehat{\mathit{Safe}}?\,([\![\; \texttt{Dealloc}\,(y_i)\;]\!]) \;=\; \left(\begin{array}{l} \widehat{\rho}'[y_i] \cap I = \emptyset \\ \wedge \widehat{\rho}'[y_i] \cap R = \emptyset \\ \wedge \bigwedge_{y_j \neq y_i} \widehat{\rho}'[y_i] \cap \widehat{\rho}'[y_j] = \emptyset \end{array}\right) \tag{4.6}$$

Note that instead of testing for object ol not being in sets I and R, we now must test that none of the labels in the value of y_i are in sets I or R. Also, we must test for pairwise disjointness of the values to be deallocated rather than testing for pairwise inequality of object labels.

In Chapter 5, we develop algorithms for verifying and inserting object deallocation commands. In that chapter we see how all of the necessary values are computed using the abstract interpreter.

4.2 Abstracting the Semantic Domains

The abstract interpreter is supposed to allow us to compute or approximate the value of a useful property of a program. We are interested in knowing which objects can be reached from each of the variables in the program.

4.2.1 Abstract Domains

Figure 4.1 contains the definition of the domains used by the abstract interpreter. We describe these domains in more detail in the remainder of this section.

Activation Labels

We summarize all activation labels in the standard domain of activation labels by the empty activation label ϵ. This abstraction of activation labels is the most extreme way of ensuring a finite domain. In Chapter 9 we investigate more precise abstractions of this domain.

The domain L is the set of static labels attached to expressions in a program. This domain is finite; its size is determined by the number of `MakeTuple` expressions appearing in the program.

Object Labels

Abstract object labels are composed of an abstract activation label and a static expression label. Since both the AL and L domains are finite, the domain of object labels OL must also be finite.

Under abstract interpretation, a variable may have a set of objects to which it may be bound because execution of an expression in different contexts may bind variables to different object. Thus, object references must be sets of object labels.

Denotable Values

The domains N and B of integers and booleans have been compressed to a single element each because we are uninterested in the actual values computed — only in the shape and connectedness of the values computed.

Values are either scalars, *e.g.*, integers or booleans, or references to aggregates, *e.g.*, tuples. An aggregate value consists of a reference to the tuple and a store containing the value of the aggregate. A reference consists of a set of object labels ls. The domain V of denotable values therefore consists of the sum of abstract integers, booleans and sets of object labels, all lifted over a bottom element $\perp$. Note that no objects are reachable from $\perp$.

Stores

Stores map individual labels to tuples. Location ol being unbound in a store σ is the same as having ol bound to $\perp$ in σ. In the abstract semantics, we use sets of labels ls as references to an object. We dereference such a set of labels as follows:

$$\langle_{Tuple}\ c_1, \cdots, c_n\rangle = \bigsqcup_{ol \in ls} \sigma[ol]$$

$n \in N$	$=$	$\underline{N}$	Numbers
$b \in B$	$=$	$\underline{B}$	Booleans
$\alpha \in AL$	$=$	ϵ	Abstract Activation Labels
$l \in L$	$=$	$\{l_0, l_1, l_2, \cdots\}$	Expression Labels
$ol \in OL$	$=$	$AL : L$	Object Labels
$ls \in Ls$	$=$	$\mathcal{P}(OL)$	Object Label Sets
$v \in V$	$=$	$(N + N + Ls)_\perp$	Denotable Values
$v_{tuple} \in Tuple$	$=$	$\langle_{Tuple}\ V, \cdots, V\rangle$	Tuples
$\sigma \in Store$	$=$	$OL \rightarrow Tuple$	Stores

Figure 4.1: Abstract value domains

$$\begin{aligned}
Abs(\perp) &= \perp \\
Abs_{AL}(\alpha) &= \epsilon \\
Abs_{OL}(\alpha : l) &= \epsilon : l \\
Abs_{LS}(ls) &= \bigcup_{ol \in ls} \{Abs_{OL}(ol)\} \\
Abs_V(v) &= \begin{cases} \perp & \text{if } v = \perp \\ \underline{N} & \text{if } v \text{ is a number} \\ \underline{B} & \text{if } v \text{ is a boolean} \\ \{Abs_{OL}(v)\} & \text{if } v \text{ is a location} \\ \top & \text{otherwise} \end{cases} \\
Abs_{Tuple}(\langle_{Tuple}\ v_1, \cdots, v_{n_1}\rangle) &= \langle_{Tuple}\ Abs_V(v_1), \cdots, Abs_V(v_{n_1})\rangle \\
Abs_{Store}(\sigma) &= \bigsqcup_{ol \in OL} \perp_{Store}[Abs_{OL}(ol) \rightarrow Abs_{Tuple}(\sigma[ol])]
\end{aligned}$$

Additional abstraction operators we require are defined below:

$$\begin{aligned}
Abs_{AEV}(\Delta^+) &= \emptyset \\
Abs_{DEV}(\Delta^-) &= \{Abs_{OL}(ol)\,|\forall \langle\ ol,\ \alpha\ \rangle \in \Delta^-\ \} \\
Abs_{REV}(\Delta^R) &= \emptyset
\end{aligned}$$

Figure 4.2: Definition of the abstraction functions

We are determining the tuple to which store σ maps the object labels in ls by taking the least upper bound of the tuples to which σ maps each label in ls.

Please remember that denotable values that are object references, or labels, are meaningless without an associated store. Although the labels themselves are very important in this semantics, the true meaning of a denotable value is tied to the object in the store named by the value's set of labels. Similarly, the set of labels of objects allocated and deallocated only has any meaning when accompanied by a store in which the allocated and deallocated objects reside.

Abstraction Functions

$$
\begin{aligned}
v \sqcup \bot &= v \\
\bot \sqcup v &= v
\end{aligned}
$$

$$ls_1 \sqcup_{Ls} ls_2 = ls_1 \cup ls_2$$

$$
v_1 \sqcup_V v_2 = \begin{cases} \underline{N} & \text{if } v_1, v_2 \text{ are both numbers} \\ \underline{B} & \text{if } v_1, v_2 \text{ are both booleans} \\ v_1 \cup v_2 & \text{if } v_1, v_2 \text{ are both Ls} \\ \top & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
&\langle_{Tuple}\ v_1, \cdots, v_{n_1}\rangle \sqcup_{Tuple} \langle_{Tuple}\ w_1, \cdots, w_{n_2}\rangle = \\
&\qquad \begin{cases} \langle_{Tuple}\ (v_1 \sqcup_V w_1), \cdots, (v_n \sqcup_V w_n)\rangle & \text{if } n_1 = n_2 \\ \top & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\sigma_1 \sqcup_{Store} \sigma_2 = \lambda ol. \begin{cases} \sigma_1[ol] \sqcup_{Tuple} \sigma_2[ol] & \text{if } ol \in OL \\ \bot & \text{otherwise} \end{cases}
$$

Figure 4.3: Least upper bound operators on value domains

Figure 4.2 contains the definitions of the abstraction functions that map values in the standard domains to values in the abstract domains. We need these functions in order to show the correctness of the abstract interpreter.

4.2.2 Least Upper Bound Operators

Figure 4.3 contains the definitions of the least-upper-bound operators on the abstract domains. The domains are all naturally ordered.

4.2.3 Reachability

The abstract interpretation of a program yields a model of what objects are created and what objects are reachable from the bindings of a `letrec` block. Because the abstract interpreter summarizes information about all executions of an expression, we must represent references to objects as sets of abstract object labels. To restate the invariant on abstract reachability, we say that if variable `x` is bound to a set of locations ls then, in any given execution, `x` can be bound to no other locations. This reachability invariant is a constraint on the structure of the abstract object label domain. The abstraction function Abs_{OL} that maps object labels to abstract object labels must enable us to preserve this constraint.

We need a precise notion of reachable objects in the abstract domains. Given a denotable value and a store, we must be able to determine which objects are reachable from that value and store.

Definition 4.1 (Abstract Object Reachability) *$\widehat{Reachable}(v, \sigma)$, the set of labels reachable from value v in store σ, is defined as follows:*

$$
\begin{aligned}
\widehat{Reachable}(\perp, \sigma) &= \emptyset \\
\widehat{Reachable}(\underline{N}, \sigma) &= \emptyset \\
\widehat{Reachable}(\underline{B}, \sigma) &= \emptyset \\
\widehat{Reachable}(ls, \sigma) &= ls \cup \left(\bigcup_{ol \in ls} SV\widehat{Reachable}(\sigma[ls], \sigma) \right)
\end{aligned}
$$

$$
\begin{aligned}
SV\widehat{Reachable}(\perp, \sigma) &= \emptyset \\
SV\widehat{Reachable}(\langle_{Tuple}\ v_1, \cdots, v_n \rangle, \sigma) &= \bigcup_i \widehat{Reachable}(v_i, \sigma)
\end{aligned}
$$

4.2.4 Ordering Operators on Domains

Figure 4.4 contains the definitions of the ordering operators for each of the abstract domains. All of the domains are naturally ordered. These operators are necessary to show correctness and termination of the abstract interpreter.

The domain orderings on labels are by name. We consider the set $\{l_0\}$ to be less than $\{l_0, l_1\}$ regardless of what those locations may be bound to in a given store.

We say store σ_1 is less than store σ_2 if, for all labels ol in the universe of object labels OL, the tuple to which ol is bound in σ_1 is less than the tuple to which ol is bound in σ_2. Tuples are compared element-by-element using the value ordering described above. Again, sets of labels are ordered by name, not by the values to which they may refer.

4.3 Abstracting the Interpreter

In the abstract interpreter, we cannot evaluate procedure calls by unfolding the body of the called procedure because this would never terminate if any of the procedures were recursive. Instead, the abstract interpreter constructs an input-output mapping for each procedure in a program. This

$$\perp \quad \sqsubseteq \quad v \quad \forall v$$

$$ls_1 \sqsubseteq_{Ls} ls_2 \quad \equiv \quad ls_1 \subseteq ls_2$$

$$v_1 \sqsubseteq_V v_2 \quad \equiv \quad \begin{cases} True & \text{if } v_1 = \perp \\ True & \text{if } v_1, v_2 \text{ are both numbers} \\ True & \text{if } v_1, v_2 \text{ are both booleans} \\ v_1 \subseteq v_2 & \text{if } v_1, v_2 \text{ are both in Ls} \\ False & \text{otherwise} \end{cases}$$

$$\sigma_1 \sqsubseteq_{Store} \sigma_2 \quad \equiv \quad \bigwedge_{l_i \in OL} \sigma_1[l_i] \sqsubseteq_{Tuple} \sigma_2[l_i]$$

$$\texttt{where}$$
$$\langle_{Tuple}\ v_1, \cdots, v_{n_1}\rangle \sqsubseteq_{Tuple} \langle_{Tuple}\ w_1, \cdots, w_{n_2}\rangle \equiv$$
$$\begin{cases} \bigwedge_i (v_i \sqsubseteq_V w_i) & \text{if } n_1 = n_2 \\ False & \text{otherwise} \end{cases}$$

Figure 4.4: Ordering operators on domains

mapping describes the behavior of a procedure over each possible set of inputs. We stress that the function mapping only approximately describes the behavior of the function.

When we abstract the interpreter, we make a major change to the clause that evaluates procedure applications so that it looks up the result of a procedure application in the input-output mapping corresponding to the procedure being applied. The job of the program evaluator, the procedure that interprets programs, is to compute the input-output mappings for each procedure. The program evaluator iterates a function that improves the approximation of the input-output mappings of each procedure until this iteration reaches fixpoint. We describe this process in the remainder of this section.

4.3.1 Computation of Input-Output Mappings

A KID$^-$ program can be viewed as a set of recursive function definitions. These definitions may be viewed as a set of equations defining the values of the functions, where the value of a function f_i is a mapping from values in the domain of f_i to values in the range of f_i. A typical system of function definition equations is shown below.

$$\begin{array}{rcl} f_1(x_1, \cdots, x_n) & = & \cdots f_i(\cdots) \cdots \\ \vdots & \vdots & \vdots \\ f_m(x_1, \cdots, x_n) & = & \cdots f_i(\cdots) \cdots \end{array}$$

If the system of equations is monotonic with respect to the values of the functions, and the heights of all chains in the domains of the functions are bounded, then we can solve this system of equations by using fixpoint iteration. We start with an initial approximation to the solution and generate successively improved approximations until we reach an approximation equal to its improved approximation — this is the exact solution to the system of equations.

We start the fixpoint iteration by using an initial approximation of each function that returns bottom for all input values.

$$\begin{array}{rcl} f_1^0(x_1, \cdots, x_n) & = & \bot \\ \vdots & \vdots & \vdots \\ f_m^0(x_1, \cdots, x_n) & = & \bot \end{array}$$

We can use bottom as the initial approximation to function f_i because it is a safe approximation to the behavior of unfolding each function application zero times. In general, the value of f_i^k is a safe approximation of the behavior of each function unfolded k times, even though it might not be a safe approximation of unfolding each function $k+1$ times. The value of f_i^∞, however, is a safe approximation to the behavior of function f_i over any depth of unfolding.

At the $k+1$th step in the fixpoint iteration, we substitute the kth approximation to function f_i, f_i^k, for each use of f_i in each equation. The substitution yields the $k+1$th approximation to the functions, as shown below:

$$\begin{array}{rcl} f_1^{k+1}(x_1, \cdots, x_n) & = & \cdots f_i^k(\cdots) \cdots \\ \vdots & \vdots & \vdots \\ f_m^{k+1}(x_1, \cdots, x_n) & = & \cdots f_i^k(\cdots) \cdots \end{array}$$

Fixpoint iteration terminates when $f_i^{k+1} = f_i^k$ for all functions f_i and all possible input values. It is guaranteed to terminate when the domains and ranges of the functions are all finite and all the functions are monotonic.

We can view this process as finding the solution to the following equation:

$$\langle f_1, \cdots, f_n \rangle \quad = \quad Y(F)$$

where F is the function that takes an approximation of each of the functions f_i and returns a refined approximation to each of the functions, and Y is the least fixpoint operator.

4.3.2 Finiteness of the KID$^-$ Abstract Domains

Although the domain of tuples is not finite, the domains of any particular function must be finite because the functions are strongly typed (monomorphically typed). The set of labels L is finite because programs are finite, and the depth of nesting of tuples passed as an argument to a function depends on the type of that function. The same is true of the result of each function — the depth of nesting of the tuples returned and the size of the sets of labels returned are both finite. Therefore, the fixpoint iteration described above must terminate. The solution to the recursive set of equations exists because the fixpoint iteration described above must terminate.

Representing Values of Functions

In the KID$^-$ abstract interpreter, function values are represented by mappings from products of denotable values and a store to pairs consisting of a denotable value and a store. The signature of these mappings is

$$Fcn = (V^* \times Store) \rightarrow (V \times Store)$$

These mappings can be thought of as a table, or set of tuples, consisting of input values and the corresponding output values.

Let us consider an example function, `f`, whose type and function mapping type are given below. If f has type $\langle_{Tuple}\ N, \langle_{Tuple}\ N, N\rangle\rangle \rightarrow N$, then the Fcn mapping associated with procedure f will have type $(V \times Store) \rightarrow (V \times Store)$.

$$\begin{array}{rcl} f & : & \langle_{Tuple}\ N, \langle_{Tuple}\ N, N\rangle\rangle \rightarrow N \\ f_{Mapping} & : & (V \times Store) \rightarrow (V \times Store) \end{array}$$

Assume that there are only two locations, l_0, of type $\langle_{Tuple}\ N, \langle_{Tuple}\ N, N\rangle\rangle$ and l_1, of type $\langle_{Tuple}\ N, N\rangle$:

$$L \quad = \quad \{l_0^{\langle_{Tuple}\ N, \langle_{Tuple}\ N,N\rangle\rangle}, l_1^{\langle_{Tuple}\ N,N\rangle}\}$$

Given this knowledge, we can enumerate all values in the domain of f:

$$\begin{array}{c} \emptyset \\ \{l_0\} \end{array} \times \perp_{Store} \left[\begin{array}{lllclll} l_0 & \rightarrow & \perp & & l_1 & \rightarrow & \perp \\ l_0 & \rightarrow & \langle_{Tuple}\ \perp, \perp\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \perp, \perp\rangle \\ l_0 & \rightarrow & \langle_{Tuple}\ \perp, \{l_1\}\rangle & \times & l_1 & \rightarrow & \langle_{Tuple}\ \underline{N}, \perp\rangle \\ l_0 & \rightarrow & \langle_{Tuple}\ \underline{N}, \perp\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \perp, \underline{N}\rangle \\ l_0 & \rightarrow & \langle_{Tuple}\ \underline{N}, \{l_1\}\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \underline{N}, \underline{N}\rangle \end{array} \right]$$

The '$\times$' signs should be read as the cross product of the possibilities for the three portions of the input domain: the value, the binding of label l_0 in the store, and the binding of label l_1 in the store. For example, the least defined element in the domain of f's mapping is:

$$\langle\ \emptyset,\ \perp_{Store}\ \rangle$$

and the most defined element is:

$$\langle\ \{l_0\},\ \perp_{Store}[l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \{l_1\}\rangle, l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle$$

Here is the range of possible results returned by function `f`:

$$\begin{array}{c} \perp \\ \underline{N} \end{array} \times \perp_{Store} \left[\begin{array}{lllclll} l_0 & \rightarrow & \perp & & l_1 & \rightarrow & \perp \\ l_0 & \rightarrow & \langle_{Tuple}\ \perp, \perp\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \perp, \perp\rangle \\ l_0 & \rightarrow & \langle_{Tuple}\ \perp, \{l_1\}\rangle & \times & l_1 & \rightarrow & \langle_{Tuple}\ \underline{N}, \perp\rangle \\ l_0 & \rightarrow & \langle_{Tuple}\ \underline{N}, \perp\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \perp, \underline{N}\rangle \\ l_0 & \rightarrow & \langle_{Tuple}\ \underline{N}, \{l_1\}\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \underline{N}, \underline{N}\rangle \end{array} \right]$$

Note that because f is an extensive function with respect to the store, the result store must contain the input store. For example, if f was applied to the store:

$$\perp_{Store}[l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \perp\rangle, l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]$$

```
{ def foo(t) =
    { a = Select₁(t);
      b = Select₂(t);
      p = (a == 5);
      r = if p then b
          else { t' = ˡ⁰MakeTuple(5,7);
                 v = ᵏ⁰foo(t');
                in v }
    in r }
  def f₀ () =
    { t0 = ˡ¹MakeTuple(3,4)
      result =  ᵏ¹foo(t0)
    in result }
}
```

Figure 4.5: A recursive example

then the only possible result stores are:

$$\perp_{Store}[l_0 \to \langle_{Tuple}\ \underline{N}, \perp\rangle, l_1 \to \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]$$

and

$$\perp_{Store}[l_0 \to \langle_{Tuple}\ \underline{N}, \{l_1\}\rangle, l_1 \to \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]$$

because all other stores in the range of f are less than or incomparable to the input store.

Computation of Function Mapping for an Example

Now let us go through the steps of constructing a mapping for the recursive example described in the previous section. Figure 4.5 contains a program consisting of a recursive procedure foo and a call to foo from the main expression of the program. First, let us examine the domain of foo and the type of the mapping we will construct for foo. Then we will work informally through the steps of building the mapping. Because this program diverges if we try to unfold the procedure calls each time the interpreter encounters a procedure application, we have to compute the fixpoint of the function that takes the initial input-output mapping of the function (the empty function mapping) and produces the final function mapping.

Figure 4.6 contains the domains of foo. Let us walk through the computation of the function mapping for foo, step by step. We start by computing the value of foo on its least defined input, and iterate until we reach fixpoint.

We only compute the value of foo on an input value if that value arises during the abstract interpretation of the program. This set of values is what we consider the *interesting* portion of the domain of foo. During each iteration we compute a new approximation of the mapping of foo, and we keep track of all values to which foo has been applied.

In order to compute the function mapping for foo, we start with an initial approximation that maps all inputs to bottom, then we use our current approximation to compute successively improved approximations until our approximation does not change.

$$
\begin{aligned}
\texttt{foo} &: (N \times N) \rightarrow N \\
\texttt{foo}_{Mapping} &: (V \times Store) \rightarrow (V \times Store) \\
L &= \{l_0^{N \times N}, l_1^{N \times N}\}
\end{aligned}
$$

$$
\begin{array}{c}
\emptyset \\ \{l_0\} \\ \{l_1\} \\ \{l_0, l_1\}
\end{array}
\times
\left[
\begin{array}{lllclll}
l_0 & \rightarrow & \perp & & l_1 & \rightarrow & \perp \\
l_0 & \rightarrow & \langle_{Tuple}\ \perp, \perp\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \perp, \perp\rangle \\
l_0 & \rightarrow & \langle_{Tuple}\ \perp, \underline{N}\rangle & \times & l_1 & \rightarrow & \langle_{Tuple}\ \underline{N}, \perp\rangle \\
l_0 & \rightarrow & \langle_{Tuple}\ \underline{N}, \perp\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \perp, \underline{N}\rangle \\
l_0 & \rightarrow & \langle_{Tuple}\ \underline{N}, \underline{N}\rangle & & l_1 & \rightarrow & \langle_{Tuple}\ \underline{N}, \underline{N}\rangle
\end{array}
\right]
$$

Figure 4.6: Domain of a function `foo`

To compute a better approximation, we evaluate the body of procedure `foo` with each set of inputs that appears in the current approximation of the mapping to compute new output values. We record the new output values in the improved approximation of `foo`'s mapping. As we evaluate the body of `foo`, if we encounter applications of `foo` to input values that do not occur in the interesting domain, we add these values to the set of values in the interesting domain.

The initial approximation for `foo` returns $\perp$ for all input values. We can then initiate the abstract interpretation of `foo` by evaluating the body of the main procedure f_0. We encounter a call with arguments:

$$\langle\ \{l_1\},\ [l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle$$

The result from this application is approximated by $\perp$, and we add this value to the interesting domain of `foo`.

To compute our next approximation to the mapping for `foo`, we evaluate its body on the single value in the interesting domain. We get the following mapping:

$$
\left[
\begin{array}{l}
\langle\ \{l_1\}, [l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle \rightarrow \\
\qquad\qquad \langle\ \underline{N}, [l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle, l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle
\end{array}
\right]
$$

and we add the input value

$$\langle\ \{l_0\},\ \perp_{Store}[l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle, l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle$$

to `foo`'s interesting domain, because a call to `foo` with these input values was encountered during the computation of the previous approximation.

And after one more iteration we reach the following approximation for `foo`:

$$
\left[
\begin{array}{l}
\langle\ \{l_0\}, [l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle, l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle \rightarrow \\
\qquad \langle\ \underline{N}, [l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle \\
\langle\ \{l_1\}, [l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle \rightarrow \\
\qquad \langle\ \underline{N}, [l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle, l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\ \rangle
\end{array}
\right]
$$

One more iteration yields the same value. Since we are not adding more entries to the interesting portion of the domain, and the values of each of the mapping entries have not changed, we have reached the fixpoint for `foo` projected onto the domain consisting of those inputs with the bindings shown in the mapping above.

Function Environments

The abstract interpreter constructs a function environment for a program. Function environments, members of domain $FEnv$, map function names to function values. Function names are drawn from domain F, and function values are input-output mappings.

$$\Phi \in FEnv \quad = \quad F \rightarrow ((V^n \times Store) \rightarrow (V \times Store))$$

Because KID$^-$ allows recursive function definitions, the interpreter must solve the set of recursive equations denoted by the program text that defines the function environment of the program.

The way we keep track of the interesting domains of each function in the program is with a domain map, or $DMAP$. Each $\mathcal{D}$ in $DMAP$ is a mapping from function names to the interesting portions of the domains of those functions. We also collect the change, or delta ($\Delta^{\mathcal{D}}$), in the interesting portion of the domain of a function.

$$\mathcal{D}, \Delta^{\mathcal{D}} \in DMAP \quad = \quad F \rightarrow \mathcal{P}(V^* \times Store)$$

The expression evaluator returns a domain map delta as one of its results.

4.3.3 Abstract Interpreter Definition

This section describes an algorithm for abstract interpretation of programs. The algorithm makes use of the fact that we are interested in only a few of the elements from the domains of the abstract functions defined in a program. This interpreter computes the function environment of a program sparsely. That is, the interpreter only computes the elements of the mapping corresponding to the input values in which we are interested and to any other inputs that are needed to compute the function environment for those interesting inputs.

The function $\mathcal{SE}_A$ takes a simple expression, and an environment, and returns the value of the expression in that environment. The function $\mathcal{E}_A$ takes an expression, an environment, a store, and a function environment, and returns the resulting value and store. The function $\mathcal{PE}_A$ takes a complete program and returns the value and store resulting from the execution of the program.

Note that the set of labels of objects allocated and deallocated during the execution of an expression or program is necessarily inexact. Under the abstract interpreter, these sets contain the abstraction of the object labels that *may* be allocated or deallocated under the standard interpreter. The most definite thing we can say is which labels were not allocated or deallocated — we cannot say that a given location was definitely allocated or deallocated. In the abstract interpreter, we do not compute the set of labels of objects that may be allocated or referenced within an expression — this is not needed to verify or insert deallocation commands. We do need to know which locations may be deallocated by an expression, however.

The following are the signatures of the semantic functions:

$$\begin{array}{l} \mathcal{SE}_A : SE \rightarrow Env \rightarrow V \\ \mathcal{E}_A : E \rightarrow Env \rightarrow Store \rightarrow FEnv \rightarrow (V \times Store \times DEVs \times DMAP) \\ \mathcal{PE}_A : Prog \rightarrow (V \times Store \times DEVs \times FEnv \times DMAP) \end{array}$$

where

$$\begin{array}{lcll} \rho \in Env & = & X \rightarrow V & \text{Environments} \\ \Delta^- \in DEVs & = & \mathcal{P}(OL \times AL) & \text{Deallocation Events} \end{array}$$

Environments, members of domain Env, map variables, X, to denotable values. The empty environment, $\perp_{Env}$, maps all variables to $\perp$. Bindings are added to an environment when we evaluate the body of a function or a `letrec` block. Domain maps and domain map deltas map function names to sets of values in the interesting domains of those functions. Abstract deallocation events track the labels of the objects that were deallocated during the interpretation of an expression.

Program Evaluator Definition

Evaluation of a program, which is performed by $\mathcal{PE}_A$, defined below, consists of computing the function environment Φ for the program, and then evaluating the main expression of the program in that function environment. The following is the definition of the program interpreter.

$$\begin{aligned}
\mathcal{PE}_A [\![\; pr \;]\!] \quad = \quad & \{ \; \langle \; \Phi_0, \mathcal{D}_0 \; \rangle = \mathit{InitialFEnv}\,(pr); \\
& \;\; \langle \; \Phi, \mathcal{D} \; \rangle \quad = \mathit{ComputeFenv}\,(pr, \Phi_0, \mathcal{D}_0); \\
& \;\; \langle \; v, \sigma, \Delta^-, \Delta^{\mathcal{D}} \; \rangle \; = \Phi[f_0][\langle \sigma \rangle]; \\
& \;\; \text{in } \left\langle v, \sigma, \Delta^-, \Phi, \Delta^{\mathcal{D}} \right\rangle \; \}
\end{aligned}$$

The abstract interpreter first constructs a function environment (Φ) that, for each function f in the program, maps particular input values of f to the result of applying f to those inputs. Whenever we encounter an application of a procedure f we fetch its input-output mapping from the incoming function environment. Then we determine the output value corresponding to the set of input values provided (including the store and activation label) and use that value as the result of the activation. We also make sure that the entry for function f and this set of inputs is non-bottom by adding these input values to the domain map for f.

Once the abstract program interpreter has computed the function environment for the program, it evaluates the body of the main procedure f_0 of the program, and returns the result of this evaluation, along with the function environment, as the result of abstract interpretation of the program.

Computing the Function Environment of a Program

The function *InitialFEnv* takes a program and returns a function environment and a domain map. The initial function environment takes each function and returns bottom. The initial domain map takes a function name and returns the set containing bottom.

$$\begin{aligned}
& \mathit{InitialFEnv}\,(pr) = \\
& \qquad \{ \; \Phi_0 \qquad = \quad \perp_{FEnv}; \\
& \qquad\;\; \forall \; f_i \in F : \\
& \qquad\;\; \mathcal{D}_0[f_i] \quad = \quad \{\perp\}; \forall \; f_i \in F \\
& \qquad \text{in } \langle \; \Phi_0, \mathcal{D}_0 \; \rangle \; \}
\end{aligned}$$

The function *ComputeFEnv*, shown in Figure 4.7, iteratively improves the approximations of the function environment and the domain map until no further information is added. It does this by computing a new entry in the function map of each function f_0 to f_k for each value in the interesting domain of the function. It also gathers new approximations to the interesting domain of the function. The process of computing new approximations is monotonic; so this is guaranteed to reach a stable value. It returns the most precise approximation as its result.

$$
\begin{array}{l}
\mathit{ComputeFEnv}\,(pr, \Phi, \mathcal{D}) = \\
\quad \{ \ [\![\ \{ \cdots f_i(x_{i,1}, \cdots, x_{i,n_i}) = e_i \cdots \} \ ]\!] = pr; \\
\quad \langle \ \Phi'_{f_0}, \mathcal{D}'_{f_0} \ \rangle = \bigsqcup\limits_{\langle v_1, \cdots, v_{n_0}, \sigma \rangle \in \mathcal{D}[f_0]} \ \{ \ \langle \ v, \sigma', \Delta^-, \Delta^{\mathcal{D}} \ \rangle = \mathcal{E}_A [\![\ e_0 \ ]\!] \perp_{Env} [v_1/x_1, \ldots, v_{n_0}/x_{n_0}] \sigma \Phi \ ; \\
\qquad\qquad \Phi = \perp_{FEnv} \left[f_0 \rightarrow \left[\begin{array}{l} \langle v_1, \cdots, v_{n_0}, \sigma \rangle \\ \quad \rightarrow \ \langle \ v, \ \sigma', \ \Delta^- \ \rangle \end{array} \right] \right]; \\
\qquad\qquad \text{in } \langle \ \Phi, \ \Delta^{\mathcal{D}} \ \rangle \} \\
\qquad \vdots \\
\quad \langle \ \Phi_{f_k}, \mathcal{D}_{f_k} \ \rangle = \bigsqcup\limits_{\langle v_1, \cdots, v_{n_k}, \sigma \rangle \in \mathcal{D}[f_k]} \ \{ \ \langle \ v, \sigma', \Delta^-, \Delta^{\mathcal{D}} \ \rangle = \mathcal{E}_A [\![\ e_k \ ]\!] \perp_{Env} [v_1/x_1, \ldots, v_{n_k}/x_{n_k}] \sigma \Phi \ ; \\
\qquad\qquad \Phi = \perp_{FEnv} \left[f_n \rightarrow \left[\begin{array}{l} \langle v_1, \cdots, v_{n_k}, \sigma \rangle \\ \quad \rightarrow \ \langle \ v, \ \sigma', \ \Delta^- \ \rangle \end{array} \right] \right]; \\
\qquad\qquad \text{in } \langle \ \Phi, \ \Delta^{\mathcal{D}} \ \rangle \} \\
\quad \Phi' = \bigsqcup\limits_{f_i} \Phi_{f_i}; \\
\quad \mathcal{D}' = \bigsqcup\limits_{f_i} \mathcal{D}_{f_i}; \\
\quad \langle \ \Phi'', \ \mathcal{D}'' \ \rangle = \mathit{if} \ \Phi' = \Phi \wedge \mathcal{D}' = \mathcal{D} \\
\qquad\qquad \mathit{then} \ \langle \ \Phi', \ \mathcal{D}' \ \rangle \\
\qquad\qquad \mathit{else} \ \mathit{ComputeFEnv}\,(pr, \Phi', \mathcal{D}'); \\
\quad \text{in } \langle \ \Phi'', \ \mathcal{D}'' \ \rangle \}
\end{array}
$$

Figure 4.7: Procedure to compute function environment

Simple Expression Evaluator Definition

The simple expression evaluator is defined in Figure 4.8. Because the integer and boolean domains have been summarized by single values, $\underline{N}$ representing any number and $\underline{B}$ representing any boolean, evaluation of constants returns less information than in the instrumented and standard interpreters. However, evaluation of variables is the same — the value of the variable is found in the current environment.

Expression Evaluator Definition

This section develops the definition of the abstracted expression evaluator. We can think of the expression evaluator as providing the rules for simplifying the right-hand-sides of the equations that define the function environment of a program.

$$\begin{array}{rcll} \mathcal{SE}_A [\![\ n \ ]\!] \rho & = & \underline{N} & \text{where } n \text{ is a number} \\ \mathcal{SE}_A [\![\ b \ ]\!] \rho & = & \underline{B} & \text{where } b \text{ is a boolean} \\ \mathcal{SE}_A [\![\ x \ ]\!] \rho & = & \rho[x] & \text{where } x \text{ is a variable} \end{array}$$

Figure 4.8: Abstracted simple expression evaluator

$$\begin{array}{rcl} \mathcal{E}_A [\![\ n \ ]\!] \rho\sigma\Phi & = & \langle \ \mathcal{SE}_A [\![\ n \ ]\!] \rho \ , \sigma, \emptyset, \bot_{DMAP} \rangle \\ \mathcal{E}_A [\![\ b \ ]\!] \rho\sigma\Phi & = & \langle \ \mathcal{SE}_A [\![\ b \ ]\!] \rho \ , \sigma, \emptyset, \bot_{DMAP} \rangle \\ \mathcal{E}_A [\![\ x \ ]\!] \rho\sigma\Phi & = & \langle \ \mathcal{SE}_A [\![\ x \ ]\!] \rho \ , \sigma, \emptyset, \bot_{DMAP} \rangle \end{array}$$

$$\mathcal{E}_A [\![\ +(se_1, se_2) \ ]\!] \rho\sigma\Phi \quad = \quad \langle \ \underline{N}, \sigma, \emptyset, \bot_{DMAP} \rangle$$

Figure 4.9: Evaluation of simple expressions and primitive operators

The first four clauses of the interpreter define the semantics of numbers, booleans, variables and arithmetic primitives. These three clauses all invoke the simple expression evaluator. These clauses are shown in Figure 4.9. The first three clauses describe how the evaluator handles simple expressions — it calls the abstract simple expression evaluator to produce the result values. The fourth clause shows how primitive arithmetic operations are interpreted — either $\underline{N}$ or $\underline{B}$ is returned, depending on the type of the operator. The evaluation of primitive arithmetic and logical operations can proceed without examining the arguments to the operator because the values of integers and booleans are ignored. These four clauses do not modify the store, so σ and $\emptyset$ are returned, and do not add any elements to the delta domain map, so $\bot_{DMAP}$ is returned.

The first major difference between the abstract expression interpreter and the instrumented interpreter is in the handling of function applications. When we interpret a procedure application in the abstract interpreter, we look up the result values in the incoming function environment rather than directly evaluating the body of the function, as we did in the standard interpreters. First, we compute the input value to function f. We use the values and current store σ as input into the function map for f. The clause of the interpreter for function applications is shown in Figure 4.10. Note that we return a delta-domain-map $\Delta^{\mathcal{D}}$ with the singleton set containing the current input value for procedure f. This ensures that we compute the value of function f applied to this input value in future iterations of *ComputeFEnv*.

The evaluation of conditionals, shown in Figure 4.11, computes a summarization of the value that the conditional could yield under any execution of the program. In the abstracted interpreter, the predicate is ignored and *both* branches of the conditional are executed. The least upper bound of the values returned by the conditional branches is returned as the result of the conditional expression.

Abstract evaluation of block expressions is nearly the same as instrumented evaluation of block expressions. Figure 4.12 shows the clause of the abstract interpreter for block expressions.

The abstract evaluation rules for tuple primitives are shown in Figure 4.13. The evaluation of the `MakeTuple` primitive is similar to the instrumented interpreter clause, except that a singleton set containing the abstract object label is returned as the result. Note that the new abstract object label is just $\epsilon : l$, the label on the `MakeTuple` primitive.

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^k f(se_1, \cdots, se_n) \ ]\!] \rho\sigma\Phi \ = \\
\quad \{ \ v_1 \qquad\qquad\qquad = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \ ; \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\quad\ \ v_n \qquad\qquad\qquad = \mathcal{SE}_A [\![\ se_n \ ]\!] \rho \ ; \\
\quad\ \ \langle \ v', \ \sigma', \ \Delta^{-\prime} \ \rangle = \Phi[f][v_1, \cdots, v_n, \sigma]; \\
\quad\ \ \Delta^{\mathcal{D}} \qquad\qquad\quad = \bot_{DMAP}[f \rightarrow \{\langle v_1, \cdots, v_n, \sigma \rangle\}]; \\
\quad\ \ \text{in } \langle \ v', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}} \ \rangle \ \}
\end{array}
$$

Figure 4.10: Abstract evaluation of function applications

$$
\begin{array}{lcl}
\mathcal{E}_A [\![\ \texttt{if}\,(se_0, e_1, e_2) \ ]\!] \rho\sigma\Phi & = & \{ \ \langle \ v_1, \sigma_1, \Delta^-{}_1, \Delta^{\mathcal{D}}{}_1 \ \rangle = \mathcal{E}_A [\![\ e_1 \ ]\!] \rho\sigma\Phi \ ; \\
 & & \quad \langle \ v_2, \sigma_2, \Delta^-{}_2, \Delta^{\mathcal{D}}{}_2 \ \rangle = \mathcal{E}_A [\![\ e_2 \ ]\!] \rho\sigma\Phi \ ; \\
 & & \quad \text{in } \langle \ v_1 \sqcup v_2, \sigma_1 \sqcup \sigma_2, \Delta^-{}_1 \sqcup \Delta^-{}_2, \Delta^{\mathcal{D}}{}_1 \sqcup \Delta^{\mathcal{D}}{}_2 \ \rangle \ \}
\end{array}
$$

Figure 4.11: Evaluation of conditional expressions

$$
\begin{array}{l}
\mathcal{E}_A [\![\ \{ \ Bs\text{—}Ds \ \texttt{in} \ x \} \ ]\!] \rho\sigma\Phi \ = \\
\quad \{ \ [\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!] \qquad\qquad = Bs; \\
\qquad [\![\ \texttt{Dealloc}\,(y_1); \cdots; \texttt{Dealloc}\,(y_k) \ ]\!] = Ds; \\
\qquad \rho_0 \qquad\qquad\qquad\qquad\qquad\qquad\quad = \rho[\bot/x_1, \cdots, \bot/x_n]; \\
\qquad \langle \ \rho', \sigma', \Delta^-, \Delta^{\mathcal{D}} \ \rangle \qquad\qquad\qquad = \mathit{EvalBindings}_A\,(Bs, \Phi, \rho_0, \sigma); \\
\qquad \Delta^{-\prime} \qquad\qquad\qquad\qquad\qquad\quad\ = \Delta^- \sqcup \bigsqcup_{y_i} \rho'[y_i]; \\
\qquad \text{in } \langle \ \rho'[x], \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}} \ \rangle \ \}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{EvalBindings}_A\,([\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!], \Phi, \rho, \sigma) = \\
\quad \{ \ \langle \ v_1, \sigma_1, \Delta^-{}_1, \Delta^{\mathcal{D}}{}_1 \ \rangle \ = \mathcal{E}_A [\![\ e_1 \ ]\!] \rho\sigma\Phi \ ; \\
\qquad\quad \vdots \\
\qquad \langle \ v_n, \sigma_n, \Delta^-{}_n, \Delta^{\mathcal{D}}{}_n \ \rangle \ = \mathcal{E}_A [\![\ e_n \ ]\!] \rho\sigma\Phi \ ; \\
\qquad \rho' \quad = \rho[(v_1 \sqcup \rho[x_1])/x_1, \cdots, (v_n \sqcup \rho[x_n])/x_n]; \\
\qquad \sigma' \quad = \bigsqcup_i \sigma_i; \\
\qquad \Delta^{-\prime} = \bigsqcup_i \Delta^-{}_i; \\
\qquad \Delta^{\mathcal{D}\prime} = \bigsqcup_i \Delta^{\mathcal{D}}{}_i; \\
\qquad \langle \ \rho'', \sigma'', \Delta^{-\prime\prime}, \Delta^{\mathcal{D}\prime\prime} \ \rangle \ = \\
\qquad\qquad\quad \mathit{if} \ \rho' \sqsubseteq \rho \wedge \sigma' \sqsubseteq \sigma \\
\qquad\qquad\quad\ \ \mathit{then} \ \langle \ \rho', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}\prime} \ \rangle \\
\qquad\qquad\quad\ \ \mathit{else} \ \mathit{EvalBindings}_A\,([\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!], \Phi, \rho', \sigma') \\
\qquad \text{in } \langle \ \rho'', \sigma'', \Delta^{-\prime\prime}, \Delta^{\mathcal{D}\prime\prime} \ \rangle \ \}
\end{array}
$$

Figure 4.12: Evaluation of block expressions

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^l\texttt{MakeTuple}(se_1, \cdots, se_m) \ ]\!] \rho\sigma\Phi \ = \\
\qquad \{ \ v_1 \quad = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \ ; \\
\qquad\qquad\quad \vdots \\
\qquad\ \ v_m \quad = \mathcal{SE}_A [\![\ se_m \ ]\!] \rho \ ; \\
\qquad\ \ v_{tuple} = \langle_{Tuple} \ v_1, \cdots, v_m \rangle \ ; \\
\qquad\ \ ol \quad = \epsilon : l; \\
\qquad\ \ v'_{tuple} = \sigma[ol]; \\
\qquad\ \ \sigma' \quad = \sigma[ol \rightarrow (v_{tuple} \sqcup v'_{tuple})]; \\
\qquad\ \ \text{in} \ \langle \ \{ol\}, \sigma', \emptyset, \perp_{DMAP} \ \rangle \ \}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_A [\![\ \texttt{Select}_i(se) \ ]\!] \rho\sigma\Phi \ = \\
\qquad \{ \ ls \qquad\qquad\qquad = \mathcal{SE}_A [\![\ se \ ]\!] \rho \ ; \\
\qquad\ \ \langle_{Tuple} \ v_1, \cdots, v_m \rangle = \bigsqcup_{ol \in ls} \sigma[ol]; \\
\qquad\ \ \text{in} \ \langle \ v_i, \sigma, \emptyset, \perp_{DMAP} \ \rangle \ \}
\end{array}
$$

Figure 4.13: Abstract evaluation of tuple primitives

In the clause for primitive $\texttt{Select}_i$, note that we took the least upper bound of all the tuples that may be referred to by ls, and then returned the ith component of that tuple. We could also have selected the ith components of all the tuples to which the set ls refers, and then returned the least upper bound of these values. We can see that the two methods are equivalent by examining the definitions of least upper bound on values and tuples.

4.4 Soundness of the Abstracted Interpreter

This section shows that our abstract interpreter is sound.

Theorem 4.2 *The abstracted interpreter $\mathcal{E}_A$ is extensive with respect to stores.*

$$
\begin{array}{l}
\forall \rho \in Env, \sigma_0 \in Store, \Phi \in FEnv, \\
\exists \langle\, v, \sigma_1, \Delta^-, \Delta^{\mathcal{D}} \,\rangle = \mathcal{E}_A [\![\, e \,]\!] \rho \sigma_0 \Phi \\
\sigma_0 \sqsubseteq \sigma_1
\end{array}
$$

Proof:

Similar to the proof of the extensionality of the standard interpreter. ■

Theorem 4.3 *The interpreter functions $\mathcal{SE}_A$ and $\mathcal{E}_A$ are monotonic.*

$$
\begin{array}{l}
\forall se \in SE,\ \forall \rho_0, \rho_1 \in Env, \\
\rho_0 \sqsubseteq \rho_1 \Rightarrow \mathcal{SE}_A [\![\, se \,]\!] \rho_0 \sqsubseteq \mathcal{SE}_A [\![\, se \,]\!] \rho_1 \\
\forall e \in E,\ \forall \rho_0, \rho_1 \in Env,\ \forall \sigma_0, \sigma_1 \in Store,\ \forall \Phi_0, \Phi_1 \in FEnv, \\
\rho_0 \sqsubseteq \rho_1 \wedge \sigma_0 \sqsubseteq \sigma_1 \wedge \Phi_0 \sqsubseteq \Phi_1 \Rightarrow \mathcal{E}_A [\![\, e \,]\!] \rho_0 \sigma_0 \Phi_0 \sqsubseteq \mathcal{E}_A [\![\, e \,]\!] \rho_1 \sigma_1 \Phi_1
\end{array}
$$

Proof:

Similar to the proof of the monotonicity of the standard interpreter. ■

Finally, we require that the abstract interpreter always terminates in a finite amount of time.

Theorem 4.4 *The abstract program evaluator $\mathcal{PE}_A$ always terminates in a finite amount of time.*

Proof:

The simple evaluator $\mathcal{SE}_A$ terminates because it either returns the value of a literal constant or else looks up a variable in the environment.

We use structural induction to show that the expression evaluator $\mathcal{E}_A$ terminates in a finite amount of time. The expression evaluator $\mathcal{E}_A$ has three cases: function application expressions, `letrec` block expressions, and all other expressions.

- The evaluation of function applications takes a finite amount of time because the expression evaluator makes a finite number of calls to the simple expression evaluator, and then looks up the result of the function application in the function environment.
- The evaluation of `letrec` blocks consists iterating to refine the environment and store of the body of the block. Each iteration consists of evaluating a finite number of expressions, so each iteration takes a finite amount of time (using our induction hypothesis). The size of all chains in our domains are finite, so it takes only a finite number of iterations for the values of the environment and store to climb to their limits.
- The evaluation of all other expressions consists of evaluating a finite number of subexpressions and combining the results in some way. Evaluating the subexpressions and combining the results each take a finite amount of time.

Finally, we require the computation of the function environment to take a finite amount of time. Each iteration of this process consists of evaluating each function over value in the interesting portion of that function's domain. All of our value domains are bounded by program size, so the functions' domains must be finite. Evaluation of the body of the function uses the expression evaluator, so that must take a finite amount of time. The fixpoint iteration used to compute the function environment must also terminate in a finite number of steps because the sizes of all domains are finite.

■

4.5 Safety of the Abstracted Interpreter

In this section we show that the abstract interpreter $\mathcal{PE}_A$ preserves the behavior of the instrumented interpreter.

An abstract interpretation is considered to be safe if the abstraction of a function preserves the behavior of the concrete function.

Definition 4.5 (Abstract Interpretation Safety) *Given domains A, $\widehat{A}$, B, and $\widehat{B}$, and abstraction functions $Abs_A : A \to \widehat{A}$ and $Abs_B : B \to \widehat{B}$, we say an abstract function $\widehat{f} : \widehat{A} \to \widehat{B}$ is* safe for *concrete function $f : A \to B$ if the following condition holds:*

$$\begin{array}{l} \forall a \in A,\ \forall \widehat{a} \in \widehat{A}. \\ Abs_A(a) \sqsubseteq_A \widehat{a} Abs_B(f(a)) \sqsubseteq_B \widehat{f}(\widehat{a}) \end{array}$$

If our abstract interpreter is safe by this definition, then it must preserve object reachability.

Theorem 4.6 *$\mathcal{SE}_A$ is safe for $\mathcal{SE}$:*

$$\begin{array}{l}\forall se \in SE,\ \forall \rho \in Env,\ \forall \widehat{\rho} \in Env_A : \\ Abs_{Env}(\rho) \sqsubseteq \widehat{\rho} \Rightarrow Abs_V(\mathcal{SE}[\![\ se\ ]\!]\rho) \sqsubseteq \mathcal{SE}_A[\![\ se\ ]\!]\widehat{\rho}\end{array}$$

Proof:

By structural induction over SE:

- If se is a boolean, then $\mathcal{SE}$ returns either `True` or `False`, which abstract to $\underline{B}$, and $\mathcal{SE}_A$ returns $\underline{B}$.
- If se is a number, then $\mathcal{SE}$ returns a number, which abstracts to $\underline{N}$, and $\mathcal{SE}_A$ returns $\underline{N}$.
- If se is a variable x, then $\mathcal{SE}$ returns $\rho[x]$ and $\mathcal{SE}_A$ returns $\widehat{\rho}[x]$. By our definition of Abs_{Env},

$$Abs_{Env}(\rho) \sqsubseteq \widehat{\rho} \Rightarrow Abs_V(\rho[x]) \sqsubseteq \widehat{\rho}[x]$$

■

Theorem 4.7 *$\mathcal{E}_A$ is safe for $\mathcal{E}_I$. Given function environment Φ for program pr:*

$$\begin{array}{l}\forall e \in pr,\ \forall \alpha \in AL,\ \forall \rho \in Env,\ \forall \widehat{\rho} \in Env_A,\ \forall \sigma \in Store,\ \forall \widehat{\sigma} \in Store_A : \\ Abs_{Env}(\rho) \sqsubseteq \widehat{\rho} \wedge Abs_{Store}(\sigma) \sqsubseteq \widehat{\sigma} \Rightarrow Abs(\mathcal{E}_I[\![\ e\ ]\!]\rho\sigma\alpha) \sqsubseteq \mathcal{E}_A[\![\ e\ ]\!]\widehat{\rho}\widehat{\sigma}\Phi\end{array}$$

Proof:

By structural induction over E:

- If e is a simple expression, then $\mathcal{E}_I$ and $\mathcal{E}_A$ call the corresponding simple evaluators, so $\mathcal{E}_A$ is safe for $\mathcal{E}_I$.
- If e is a primitive arithmetic expression, then $\mathcal{E}_A$ returns either $\underline{N}$ or $\underline{B}$, depending on its type. These values contain the abstraction of all possible values that $\mathcal{E}_I$ could return.
- If e is a function application, then $\mathcal{E}_A$ calls $\mathcal{SE}_A$ to evaluate the arguments. These abstract argument values contain the abstractions of the corresponding calls to $\mathcal{SE}_I$ made by $\mathcal{E}_I$. The function environment for a program maps a set of abstract inputs to the most general value a function could return when applied to any concrete inputs contained in the abstract inputs. Therefore, $\mathcal{E}_A$ is safe for $\mathcal{E}_I$ for function applications because it looks up the result in the function environment.
- If e is a conditional, then $\mathcal{E}_A$ returns the least upper bound of the evaluation of both branches of the conditional. This value is greater than the result of evaluating either of the branches of the conditional, so it must contain the value of $\mathcal{E}_I$ applied to the branch of the conditional that is taken under the instrumented interpreter.
- If e is a `letrec` block, then evaluation consists of fixpoint iteration to compute the block's environment and store. For each iteration, we take an approximation of the environment and store and generate refined approximations. This process is safe, by our induction hypothesis, because we call $\mathcal{E}_A$ and $\mathcal{E}_I$ on the subexpressions to compute the contributions to the new approximations to the environment and store. The final result must be safe, because each iteration is safe and because both $\mathcal{E}_A$ and $\mathcal{E}_I$ are monotonic.

- If e is a tuple primitive, then the simple expression evaluators are called to evaluate the arguments and a tuple is constructed or dereferenced. The object label constructed by $\mathcal{E}_A$ is an abstraction of the object label constructed by $\mathcal{E}_I$. Any tuple allocated by $\mathcal{E}_A$ is an abstraction of a tuple allocate by $\mathcal{E}_I$ because each of the components under $\mathcal{E}_A$ is an abstraction of the corresponding values under $\mathcal{E}_I$.

■

Theorem 4.8 *$\mathcal{PE}_A$ is safe for $\mathcal{PE}_I$.*

$$\begin{array}{l}\forall pr \in Prog, \\ Abs\,(\mathcal{PE}_I [\![\ pr\ ]\!]\,) \sqsubseteq \mathcal{PE}_A [\![\ pr\ ]\!]\end{array}$$

Proof:

In order to show that the abstract program interpreter is safe for the instrumented program interpreter, we must show that the abstract interpreter constructs a function environment that is safe with respect to the behavior of each of the functions in the program.

We do this by induction on the depth of nesting of function calls.

- Base case: The initial function environment Φ^0 maps all functions to input-output tables that map all inputs to bottom. Φ^0 is safe for all expressions that do not call procedures.
- Induction Hypothesis: We assume that function environment Φ^k is safe for the abstract interpretation of expressions that expand to a call depth of k. To compute the value of the Φ^k, the $k+1$st approximation to the function environment, we evaluate the body of each function applied to each value in the interesting domain of the function using $\mathcal{E}_A$ and function environment Φ^k. This yields Φ^{k+1} that is safe for expressions that expand to a call depth of $k+1$, because $\mathcal{E}_A$ is safe for $\mathcal{E}_I$.

■

4.6 Determining Object Lifetimes Statically

We can apply the abstract interpreter to a program *prog* to get a value, a store, and a set of deallocation events. Consider the example shown in Figure 4.14. The result of evaluating the program is a number and a store containing one tuple. If we follow the execution of the application of `f` to 68, we see that its result is

$$\begin{array}{ll}\langle & \underline{N}, \\ & \bot_{Store}[\epsilon : l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle], \\ & \emptyset, \\ & \Delta^{\mathcal{D}}\ \rangle\end{array}$$

meaning that a number is returned as the result and that a tuple labeled $\epsilon\!:\!l_0$ was allocated during the execution of the application. Because the tuple is not reachable from the result we know that the lifetime of the tuple ends when the invocation of `f` ends. With a little more information about which identifiers in `f` are bound to the tuple, we could transform the program so that the storage associated with the tuple is reclaimed when `f` terminates.

```
{ def f(w) =
    { t = k0g(w);
      a = Select1(t);
      b = Select2(t);
      r = (a * b);
     in r };
  def g(x) =
    { y = (x-21);
      t = l0MakeTuple(x,y) ;
    in t }
  def f0() =
    k1f(68)
};
```

Figure 4.14: Example with non-nested structures

```
{ def f(w) =
    { t1 = k0g(w);
      w2 = w * 2;
      t2 = k1g(w2);
      r = (w * w2);
      t3 = l1MakeTuple(t1,t2,r);
     in t3 };
  def g(x) =
    { y = (x-21);
      t = l0MakeTuple(x,y) ;
    in t };
  def f0() =
    k2f(68);
};
```

Figure 4.15: Example with false sharing

The example shown earlier in this chapter in Figure 4.5 is slightly more complicated than the one we just did. It consists of a recursive procedure, `foo`, that allocates a tuple in each recursive iteration. We went through the steps of abstract interpretation in detail in Section 4.3.2, obtaining the following value:

$$\begin{array}{ll} \langle & \underline{N}, \\ & \perp_{Store}[\epsilon : l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle, \epsilon : l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle], \\ & \emptyset, \\ & \Delta^{\mathcal{D}} \ \rangle \end{array}$$

One thing we can conclude by examining the program and this result is that both tuples allocated are no longer in use at the end of the program, because the program returns a number as its result.

The example in Figure 4.15 is interesting because it shows how the abstraction of labels can cause apparent sharing of structures, when in the standard or instrumented semantics there actually would be no sharing. The result of this program under abstract interpretation is:

$$\begin{array}{ll}\langle & \{l_1\}, \\ & \perp_{Store}\left[\begin{array}{l} l_0 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N}\rangle \\ l_1 \rightarrow \langle_{Tuple}\ \{l_0\}, \{l_0\}, \underline{N}\rangle \end{array}\right], \\ & \emptyset, \\ & \Delta^{\mathcal{D}}, \qquad\qquad\qquad\qquad\qquad\qquad \rangle \end{array}$$

That is, the result of the program is the structure contained in locations $\{l_1\}$, which is a three tuple containing a reference to location l_0, another reference to location l_0, and a number. Note that while the tuples allocated by the two calls to g would have been allocated in different locations $\epsilon.k_2.k_0\!:\!l_0$ and $\epsilon.k_2.k_1\!:\!l_0$ under the instrumented semantics, they have been assigned the same location under the abstract semantics. Thus, any analysis performed using this abstraction of activation labels will not be able to distinguish the tuples allocated by distinct calls to a procedure. Chapter 6 discusses an improved approximation of activation labels that solves this problem.

Chapter 5 presents two algorithms that use the abstract interpreter defined in this chapter. The first algorithm verifies the safety of deallocation commands in programs, and the second inserts deallocation commands into programs. The basic approach is similar to what we have done in this section. The compiler uses the abstract interpreter to compute input-output mappings of all procedures. Then the compiler processes the body of each procedure, computing the possible bindings of all variables in the body of the procedure and using our deallocation safety criteria to verify or insert deallocation commands.

Chapter 5

Verifying and Inserting Deallocation Commands

We have seen how to interpret KID^{-} programs in such a way as to determine what objects are created by the program (or each procedure in the program) and what objects passed into a procedure are reachable from the result of that procedure. We now investigate how to turn our abstract interpreter into an algorithm to verify deallocation commands and an algorithm to insert deallocation commands.

The verification and insertion algorithms compute the function environment for a whole program and then operate on each procedure of the program. Both algorithms compute the function environment for the program and then recursively traverse the body of each procedure, calling the abstract expression evaluator $\mathcal{E}_A$ to provide a summary of the value to which each identifier could be bound.

Both the verification and insertion algorithms must analyze procedure bodies with respect to a set of arguments provided to that procedure. In this chapter we discuss how the choice of input values affects the performance of the analysis of procedure bodies. We also describe how we choose input values for use in the verification and algorithms.

As we did in Chapter 4, we restrict our discussion in this chapter to programs using tuples as their only data structure. In the next two chapters we discuss both the additions to the abstract interpreter, and the insertion and verification algorithms needed to handle arrays, algebraic types, and lists.

The first section of this chapter presents formal conditions for the correctness of a deallocation statement. The second section discusses the choice of input values used during the analysis of a procedure and presents an algorithm for choosing these values. The third section presents a mechanical algorithm for verifying the correctness of deallocation statements in KID^{-} programs. The final section of this chapter presents a simple algorithm for inserting correct deallocation statements into a program.

5.1 Object Deallocation Safety

Let us consider a deallocation statement in block expression e, shown below, and use the abstract semantics to show the conditions under which this deallocation statement can never lead to a run-time error. Expression e is a generic `letrec` expression containing several deallocation commands.

$$
\begin{array}{rl}
e = & \{\ x_1 = e_1; \\
& \quad \vdots \\
& x_n = e_n; \\
& — \\
& \texttt{Dealloc}\,(y_1) \\
& \quad \vdots \\
& \texttt{Dealloc}\,(y_m) \\
& \text{in } x_j\ \}
\end{array}
$$

where the environment, store, and function environment in which e is to be evaluated are ρ, σ, and Φ, respectively.

We can compute environment ρ' and store σ', the resulting environment and store for the block bindings, Δ^-, the set of labels deallocated by the block bindings, and v, which is the result of the evaluation of the expression, as shown below:

$$
\begin{array}{rcl}
\rho_0 & = & \rho[\perp/x_1, \cdots, \perp/x_n] \\
\langle\ \rho', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}}\ \rangle & = & \mathit{EvalBindings}_A\,(\llbracket\ x_1 = e_1; \ldots; x_n = e_n\ \rrbracket, \Phi, \rho_0, \sigma, \perp_{DMAP}) \\
v & = & \rho'[x_j] \\
R & = & \widehat{\mathit{Reachable}}\,(v, \sigma') \\
I & = & \bigcup\limits_{y \in FV\,(e)} \widehat{\mathit{Reachable}}\,(\rho'[y], \sigma)
\end{array}
$$

Consider each variable y_i whose value is deallocated in the above block expression. If the value of y_i is to be deallocated safely upon termination of the `letrec` block, then the value to which y_i is bound, $\rho'[y_i]$, must be a reference ls to a tuple. Furthermore, that tuple must have been allocated within the execution of expression e. Therefore, none of the labels to which y_i could be bound may be in the set of labels inherited from the context in which this expression is executed. Also, none of the labels in $\rho'[y_i]$ can be reachable from the result $\rho'[x_j]$ of the block expression. Finally, none of those labels can be in the set of objects deallocated by other deallocation commands.

Condition 5.1 (Deallocation Command Safety) *The deallocation statement* `Dealloc`(y_i)*, shown in the code fragment above, is* safe*, if the following three conditions hold:*

1. $\rho'[y_i] \cap I = \emptyset$ *(y_i is not inherited)*
2. $\rho'[y_i] \cap R = \emptyset$ *(y_i does not escape)*
3. $\forall y_j \neq y_i\ .\ \rho'[y_i] \cap (\rho'[y_j] \cup \Delta^-) = \emptyset$ *(y_i is not deallocated elsewhere)*

If Condition 5.1 is satisfied, then Theorem 3.10 from Chapter 3 applies and it is guaranteed that this deallocation command will not lead to dangling pointer errors.

5.2 Choice of Procedure Arguments

We need to determine the behavior of a procedure over all possible values to which the procedure could be applied in order to verify that the deallocation commands in that procedure are safe or in order to insert safe deallocation commands. One way to do this is to analyze a procedure when it is applied to the least upper bound of all the abstract values in the domain of that procedure. Another choice of input values would be the least upper bound of the values in the interesting portion of the domain of the procedure. We discuss the use of the interesting domain of the procedure in this section, discuss how it sometimes prevents us from verifying deallocation commands that are actually correct, and then develop a better set of input values for use during analysis.

5.2.1 Most General Input Values

Let us consider the analysis of the body of a procedure when applied to the least upper bound of the values in the interesting domain of that procedure. For example, let us analyze the procedure `foo`, defined below:

```
def foo (w,n) =
  { a = Select₁(w);
    b = Select₂(w);
    c = a + 1;
    p = c < n;
    r = if p then
         { w' = ˡ⁰MakeTuple(c,b);
           r' = ᵏ⁰foo(w');
         in r' };
        else b;
   in r };
```

where the interesting portion of the domain of `foo` might be:

$$\left\{ \begin{array}{l} \langle \bot, \bot, \bot_{Store} \rangle \, , \\ \langle \{l_1\}, \underline{N}, [l_1 \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle] \rangle \, , \\ \langle \{l_0\}, \underline{N}, [l_1 \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle \, , l_0 \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle] \rangle \end{array} \right\}$$

where the tuple labeled l_1 was allocated somewhere else in the program.

The least upper bound of these values is the triple:

$$\langle \{l_0, l_1\}, \underline{N}, [l_1 \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle \, , l_0 \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle] \rangle$$

If we examine this program by hand, we see that the lifetime of the object bound to `w'` is contained in the lifetime of the `letrec` block in the `then` side of the conditional, because `foo` never returns its input — so `w'` does not escape — and `w'` is always bound to a freshly allocated tuple – so `w'` is allocated within the `letrec` block.

However, if we apply the procedure `foo` to the most general input value that we constructed above, we find the following possibilities for the bindings of variables `w`, `w'` and `r'` in environment ρ' within the body of `foo`:

$$\begin{array}{rcl} \rho'[\mathtt{w}] & = & \{l_0, l_1\} \\ \rho'[\mathtt{w'}] & = & \{l_0\} \\ \rho'[\mathtt{r'}] & = & \underline{N} \end{array}$$

The set I of labels that may be inherited by procedure `foo` is the set $\{l_0, l_1\}$, and the set of labels that may escape from `foo` is empty. Thus, even though we can determine by inspection that `w'` is never bound to an object that escapes from the `letrec` block, we find that `w'` may be bound to an object inherited by procedure `foo`. We cannot determine that the lifetime of `w'` is contained in the inner `letrec` block using this approach to lifetime analysis.

In this example, we have come to a safe conclusion. It is always safe to overestimate the lifetime of an object, but if the overestimates are too large we will never be able to verify or insert any deallocation commands. In fact, if we we follow this strategy of using the most general input value we analyze a procedure, we will never be able to verify the deallocation of a structure that is created by a procedure, passed to a recursive call, but not returned from the procedure. The reason is that passing an object label to a recursive call guarantees that the label is in the most general input value; therefore, the label will always be considered inherited by the procedure.

5.2.2 Desired Properties for Input Values

What are the important properties of the input values to which we apply a procedure during analysis? From the standpoint of lifetime analysis, the most important thing we know about these values is that they came from outside the procedure, and that if some variable within the procedure may be bound to one of these values, then the lifetime of the object to which that variable is bound may not be enclosed by the lifetime of the procedure. So we desire to choose input values in such a way that we can determine which variables are "contaminated" by input values, without getting any spurious contamination signals. We must also choose input values so that we never miss any contamination signals. Therefore, we cannot use bottom as an input value when analyzing a procedure.

The input values we choose must also have the right type. We cannot apply a function to a number if it expects a tuple.

Finally, we must be able to show that analysis of the body of a function with respect to some input value yields correct values for all possible values to which the function could be applied under the standard interpreter.

5.2.3 Representative Input Values

In this section we present a method for creating *representative* input values that allow us to avoid the false contamination we found in Section 5.2.1. We show that analysis performed with respect to these input values is safe for all possible inputs, up to renaming of the inputs.

Let us analyze procedure `foo` when applied to the following representative value $\vec{v_{rep}}$:

$$\langle \{l_{-1}\}, \underline{N}, [l_{-1} \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle] \rangle$$

where label l_{-1} is a new label that does not occur anywhere in the program.

Now if we evaluate the body of `foo` and determine the bindings of `w`, `w'` and `r'` we get the following values:

$$\begin{aligned} \rho'[\texttt{w}] &= \{l_{-1}\} \\ \rho'[\texttt{w}'] &= \{l_0\} \\ \rho'[\texttt{r}'] &= \underline{N} \end{aligned}$$

The set I of labels that may be inherited by procedure `foo` is the set $\{l_{-1}\}$, and the set of labels that escapes from `foo` is empty. In this case, the lifetime of the object to which `w'` is bound is contained in the lifetime of the inner `letrec` block.

Name Invariance

The question we must now answer is whether the behavior of `foo` when applied to this input value tells us anything about the behavior of `foo` when applied to other values. If we want to determine the behavior for an input vector that contains l_i instead of l_{-1}, such as the following input vector $\vec{v}$:

$$\langle \{l_i\}, \underline{N}, [l_i \rightarrow \langle_{\,Tuple}\ \underline{N}, \underline{N}\rangle\,, \cdots]\rangle\,,$$

then we can take the bindings computed for `foo` applied to input vector $\vec{v}_{rep}$, rename all occurrences of l_{-1} to l_i, and end up with the bindings for `foo` applied to l_{-1}. For instance, we rename l_{-1} to l_i in our analysis of `foo` with respect to the representative value $v_{\vec{rep}}$, we obtain the following information about the environment in the body of `foo`:

$$\begin{array}{rcl} \rho'[\mathtt{w}] & = & \{l_i\} \\ \rho'[\mathtt{w}'] & = & \{l_0\} \\ \rho'[\mathtt{r}'] & = & \underline{N} \end{array}$$

which is exactly the result if we directly analyzed procedure `foo` applied to $\vec{v}$.

If more than one location was passed as an argument, then we substitute the set of labels for the set containing l_{-1}, and duplicate the bindings of l_{-1} in the store for each of the labels in the desired input value.

We would like to show that we can take an appropriate representative input value, analyze a function with respect to that input value, and determine safe behavior for that function applied to any input value given the behavior of the function over the representative input value. In order for the behavior of a function over its representative input value to tell us anything about the behavior of the function over other input values, the representative input value must satisfy the following three conditions:

1. The representative value must have the same type as all other input values to this function.

2. None of the values reachable from the representative input values may be bottom.

3. The labels used in the representative input value must be distinct from all labels occurring statically in the program.

The first condition is almost redundant; all values to which a function is applied must have the same type.

The second condition is required because the result of a function applied to any value that is in some way "less than" the representative input value will be less than the result of the function applied to the representative input value. If some component of the representative input value is bottom, then any input that has a non-bottom value for that component will cause different behavior.

The third condition is required because we would like to be able to perform a substitution on the result of a function applied to the representative input value in order to obtain the result of the

$$\begin{aligned}
Subst_V\,(ls, l_o, \underline{N}) &= \underline{N} \\
Subst_V\,(ls, l_o, \underline{B}) &= \underline{B} \\
Subst_V\,(ls, l_o, \{ls'\}) &= \begin{cases} (ls' - \{l_o\}) \cup ls & \text{if } l_o \in ls' \\ ls' & \text{otherwise} \end{cases}
\end{aligned}$$

$$Subst_{Ls}\,(ls, l_o, \{ls'\}) = \begin{cases} (ls' - \{l_o\}) \cup ls & \text{if } l_o \in ls' \\ ls' & \text{otherwise} \end{cases}$$

$$Subst_{Env}\,(ls, l_o, \rho) = \lambda x.\ Subst_V\,(ls, l_o, \rho[x])$$

$$Subst_{SV}\,(ls, l_o, \langle_{Tuple}\ v_1, \cdots, v_m\rangle) = \langle_{Tuple}\ Subst_V\,(ls, l_o, v_1), \cdots Subst_V\,(ls, l_o, v_m)\rangle$$

$$Subst_{Store}\,(ls, l_o, \sigma) = \lambda l. \begin{cases} Subst_{SV}\,(ls, l_o, \sigma[l_o]) & \text{if } l \in ls \\ Subst_{SV}\,(ls, l_o, \sigma[l]) & \text{otherwise} \end{cases}$$

$$Subst_{FEnv}\,(ls, l_o, \Phi) =$$

$$\bigsqcup_{f_i} \left(\bigsqcup_{\vec{v}\ s.t.\ \Phi[f_i][\vec{v}] \neq \perp} \perp_{FEnv} [f_i \rightarrow [Subst\,(ls, l_o, \vec{v}) \rightarrow Subst\,(ls, l_o, \Phi[f_i][\vec{v}])]] \right)$$

$$\begin{aligned}
&Subst^*\,(\{\langle\ ls_1,\ l_1\ \rangle, \cdots, \langle\ ls_n,\ l_n\ \rangle\}, v) = \\
&\qquad Subst^*\,(\{\langle\ ls_1,\ l_1\ \rangle, \cdots, \langle\ ls_{n-1},\ l_{n-1}\ \rangle\}, Subst\,(ls_n, l_n, v)) \\
&Subst^*\,(\emptyset, v) = v
\end{aligned}$$

Figure 5.1: Definition of procedure *Subst*

function applied to some other value. We will not get a correct result if we rename the objects allocated within the procedure call; we only want to rename or substitute for values passed as input to the function. After all, the most important thing we know about the representative input values is that they came from outside the procedure.

We define procedure *Subst* in Figure 5.1. This procedure takes a set of labels ls, a label l_o, and a value v, and substitutes ls for all occurrences of label l_o in v. We define version of *Subst* to operate on denotable values, storable values, environments, stores, and function environments.

Let $\mathcal{RIV}$ be a procedure that takes a function's type and constructs a representative input vector for that function that satisfies the three conditions given above. Let *Match* be a procedure that takes an input vector $\vec{iv}$ to a function and a function's representative input vector $\vec{riv}$, and produces a substitution θ such that

$$\vec{iv} \sqsubseteq \mathit{Subst}^*(\theta, \vec{riv}).$$

Theorem 5.2 (Name Invariance) *Given program pr and the function environment Φ for that program:*

$$\begin{aligned} \forall f_i \ \ &\in pr, \ \exists \vec{riv} = \mathcal{RIV}[\![\ \mathit{TypeOf}(f_i, pr) \ ]\!], \\ &\forall \vec{iv} \ \in \mathit{Domain}(f_i), \ \exists \theta = \mathit{Match}(\vec{iv}, \vec{riv}), \\ &\quad \vec{iv} \sqsubseteq \mathit{Subst}^*(\theta, \vec{riv}) \ \Rightarrow \ \Phi[f_i][\vec{iv}] \sqsubseteq \mathit{Subst}^*(\theta, \Phi[f_i][\vec{riv}]) \end{aligned}$$

Proof:

Sketch of Proof by Contradiction:

Let:

$$\begin{aligned} r_{\vec{iv}} &= \Phi[f_i][\vec{iv}] \\ r_{\vec{riv}} &= \Phi[f_i][\vec{riv}] \end{aligned}$$

Assume that: $\Phi[f_i][\vec{iv}] \not\sqsubseteq \mathit{Subst}^*(\theta, \Phi[f_i][\vec{riv}])$

Every portion of the results of function applications, in this case $r_{\vec{iv}}$ and $r_{\vec{riv}}$, either came from the input to the function or was created within the function.

- The representative input vector $\vec{riv}$ is at least as well defined as the input vector of interest $\vec{iv}$, and so execution of the body of the function should have proceeded at least as far in the case of $\vec{riv}$ as in the case of $\vec{iv}$. Therefore, all portions of the result should be at least as well defined. For that reason, all portions of the result that came from the input should be at least as well defined in the case of $\mathit{Subst}^*(\theta, r_{\vec{riv}})$ as in the case of $r_{\vec{iv}}$, unless some of the inputs reachable under one case were not reachable under the other. But the abstract interpreter preserves reachability, so all of the components of $\mathit{Subst}^*(\theta, r_{\vec{riv}})$ that were inherited from the input vector $\vec{riv}$ must contain the portions of the result $r_{\vec{iv}}$ that came from the input $\vec{iv}$, because input $\mathit{Subst}^*(\theta, \vec{riv})$ contained all of input $\vec{iv}$.

  Contradiction.
- Again, all code in the body of function f_i must have executed at least as far when applied to $\vec{riv}$ as it did when applied to $\vec{iv}$, because $\vec{riv}$ is more defined (has more non-bottom components). Therefore all portions of result $r_{\vec{riv}}$ that were created within the function body should be at least as well defined as the portions of result $r_{\vec{iv}}$ that were created within the function body. Furthermore, all of these values that are object labels must be the same in both cases, because all object labels depend solely on the text of the program, not the inputs to the function. Therefore, it must be the case that the portions of result $r_{\vec{riv}}$ that were created within the function must contain the portions of $r_{\vec{iv}}$ that were created withing the function, even before substitution. Furthermore, none of the labels being renamed by substitution θ are created within the body of the function — the contract of $\mathcal{RIV}$ is to use labels from outside of the program.

  Contradiction.

Both of these paths lead to contradiction, so our assumption must be false.

■

There remains the question of determining if label l_0 can ever, under the concrete interpreter, both be allocated within the inner `letrec` block and be inherited by the body of the same instance of procedure `foo`. The only way an object can be allocated within an activation of a procedure procedure and passed into the same activation of the procedure is if the object is returned as part of the result and the caller of the procedure passes that value back into the procedure as an argument. Under this condition, the object's lifetime cannot be bounded by the lifetime of the procedure, because the object escapes as part of the procedure's result. If the object is never returned as part of the result, then any object passed into the procedure with the same label as an object allocated within the procedure must be an instance of an object allocated within a different activation of the same procedure.

Theorem 5.2 has two consequences. First, it allows us to construct a representative input value for each function and analyze the function applied to that representative input in order to verify or insert deallocation commands in the body of the function. It shows us that the representative input vector is equivalent to the most general input vector in the most important way: distinguishing the values that came from outside the function from those that were created within the function. The use of representative input vectors in many cases allows us to avoid the false aliasing problem that reduces the effectiveness of the deallocation safety verification and deallocation insertion algorithms.

Second, it allows us to derive a conservative approximation of the result of a function applied to a particular input value from the result of the function applied to the representative input. Theorem 5.2 guarantees that if the representative input is chosen appropriately, then the result after substitution will be an approximation of the actual result.

Two problems remain: how to choose the representative input for a given function and how to choose substitutions.

Constructing Representative Input Values for Functions

Our approach is to generate an input value for each procedure based on its type so that if we analyze or transform the function for that input value the result will be correct for all input values. This allows us to ask more precise questions about the function because we can guarantee that there is no false aliasing between the inputs to the function and any structures it may allocate.

The type of a function is a member of the domain $FunctionType$, defined below. Argument and result types of a function are drawn from the domain $Type$.

$$\begin{aligned} FunctionType &= (Type \times \cdots \times Type) \rightarrow Type \\ \tau \in Type &= N \mid B \mid \langle_{Tuple}\ Type \times \cdots \times Type \rangle \end{aligned}$$

The function $\mathcal{RIV}$ takes a function type and returns a representative input value: a tuple of abstract values and an abstract store. This procedure makes use of function $\mathcal{CV}$, which constructs a single value-store pair from a single type. The signatures of functions $\mathcal{RIV}$ and $\mathcal{CV}$ are given below:

$$\begin{aligned} \mathcal{RIV} &: FunctionType \rightarrow (V \times \cdots \times V \times Store) \\ \mathcal{CV} &: Type \rightarrow (V \times Store) \end{aligned}$$

Finally, we need a function, *TypeOf*, which gives us the type of a procedure in the program *pr*. *TypeOf* takes a function identifier and a program and returns the type of that function.

$$\texttt{TypeOf} \;:\; F \rightarrow Prog \rightarrow FunctionType$$

The procedure $\mathcal{CV}$, in the case of a scalar type, returns $\underline{N}$ or $\underline{B}$, as appropriate, and the empty store. In the case of a tuple type, $\mathcal{CV}$ is called recursively on each of the component types, a new label l is allocated, and the set containing l is returned as the value. The resulting store is the least upper bound of each of the component stores with location l bound to the tuple of the component values.

$$\begin{array}{rcl}
\mathcal{CV}[\![\; N \;]\!] & = & \langle\; \underline{N},\ \perp_{Store} \;\rangle \\
\mathcal{CV}[\![\; B \;]\!] & = & \langle\; \underline{B},\ \perp_{Store} \;\rangle \\
\mathcal{CV}[\![\; \langle_{Tuple}\ \tau_1, \cdots, \tau_n\rangle \;]\!] & = & \{\ \langle\ v_1,\ \sigma_1\ \rangle = \mathcal{CV}[\![\; \tau_1 \;]\!]\ ; \\
 & & \qquad\qquad \vdots \\
 & & \quad \langle\ v_n,\ \sigma_n\ \rangle = \mathcal{CV}[\![\; \tau_n \;]\!]\ ; \\
 & & \quad \sigma' \qquad\quad = \bigsqcup_i \sigma_i ; \\
 & & \quad l \qquad\quad\ = Newloc\,(); \\
 & & \quad \text{in } \langle\ \{l\},\ \sigma'[l \rightarrow \langle_{Tuple}\ v_1, \cdots, v_n\rangle]\ \rangle\ \}
\end{array}$$

We call function *Newloc* to give us a label that cannot appear in the program — this guarantees that we do not get any false aliasing between the initial arguments to a function and the object labels allocated within the function.

The function $\mathcal{RIV}$ takes the tuple of function argument types and calls function $\mathcal{CV}$ to construct a value and store for each of those types. It returns a tuple containing each of the values and the least upper bound of the stores. Note that because we construct these stores with disjoint locations, the least upper bound of these stores is the same as the concatenation of the stores.

$$\begin{array}{rcl}
\mathcal{RIV}[\![\; (\tau_1 \times \cdots \times \tau_n) \rightarrow \tau_r \;]\!] & = & \{\ \langle\ v_1,\ \sigma_1\ \rangle = \mathcal{CV}[\![\; \tau_1 \;]\!]\ ; \\
 & & \qquad\qquad \vdots \\
 & & \quad \langle\ v_n,\ \sigma_n\ \rangle = \mathcal{CV}[\![\; \tau_n \;]\!]\ ; \\
 & & \quad \sigma' \qquad\quad = \bigsqcup_i \sigma_i ; \\
 & & \quad \text{in } \langle v_1, \cdots, v_n, \sigma' \rangle\ \}
\end{array}$$

5.3 An Algorithm for Verifying Deallocation Commands

This section defines $\mathcal{VP}$, an algorithm that takes a program *pr* and returns a set of the expression labels of possibly incorrect deallocation commands in the program. This algorithm errs on the conservative side, returning labels of deallocation commands that may never cause dangling pointer errors at run-time, but it never returns the empty set for programs that are incorrect.

Procedure $\mathcal{VP}$ verifies programs using monotonic reasoning: it first assumes all procedures are correct and iteratively improves this approximation until it finds all the procedures that could have dynamic errors deallocating structures. We use a new mapping — a correctness map — that gives the most up-to-date information about the correctness of each procedure. A correctness map takes a procedure name and returns either $\emptyset$ if the procedure contains only correct deallocation commands or a non-empty set of incorrect deallocation command labels otherwise.

$$\Psi \in CMAP \quad = \quad F \rightarrow \mathcal{P}(L)$$

The function $\mathcal{VE}$ verifies that all deallocation commands within an expression are correct with respect to a given environment, store, and function environment.

Here are the signatures of the verification procedures:

$$\begin{aligned} \mathcal{VP} &: \ Prog \rightarrow \mathcal{P}(L) \\ \mathcal{VE} &: \ E \rightarrow Env \rightarrow Store \rightarrow FEnv \rightarrow CMAP \rightarrow \mathcal{P}(L) \end{aligned}$$

These functions are defined in the following two sections.

5.3.1 Verification of Deallocation within a Program

The function $\mathcal{VP}$ verifies that all of the deallocation statements in the main body of the program and in each of the functions of the program cannot lead to dereferencing a deallocated location under any execution of the program. The definition of $\mathcal{VP}$ is shown below:

$$\begin{array}{l} \mathcal{VP}[\![\ pr \ ]\!] \ = \\ \quad \{ \ [\![\ \{\cdots f_i(x_1,\cdots,x_n) = e_i \cdots \ \mathtt{in} \ e\} \ ]\!] = pr; \\ \quad\ \ \langle \ \Phi_0, \mathcal{D}_0 \ \rangle = \mathit{InitialFEnv}\,(pr); \\ \quad\ \ \langle \ \Phi, \mathcal{D} \ \rangle \quad = \mathit{ComputeFEnv}\,(pr, \Phi_0, \mathcal{D}_0); \\ \\ \quad\ \ \Psi_0[f_i] \qquad = \emptyset, \quad \forall\ f_i; \\ \quad\ \ \Psi \qquad\qquad = \mathit{ComputeCMAP}\,(pr, \Phi, \mathcal{D}, \Psi_0); \\ \\ \quad\ \ ds \qquad\quad\ = \mathcal{VE}[\![\ e_0 \ ]\!] \perp_{Env} \perp_{Store} \Phi\Psi \ ; \\ \quad\ \ \text{in } ds \ \} \end{array} \tag{5.1}$$

where expression e_0 is the body of the main function $\mathtt{f}_0$. Procedure $\mathcal{VP}$ calls procedure *ComputeFEnv* to compute the function environment and the interesting domain map of the program. Then it calls *ComputeCMAP* to compute the correctness map for the program. The *CMAP* Ψ takes function names and returns the list of expression labels of deallocation commands that may be incorrect. Finally, procedure $\mathcal{VP}$ calls procedure $\mathcal{VE}$ to verify the correctness of expression e_0, the body of the main procedure f_0. If there are no incorrect deallocation commands that may be called from the main body of the program, then all deallocation commands in the program must be correct (or else unreachable from the main procedure).

We revise the function *InitialFEnv* that takes a program and returns a function environment and a domain map. The empty function environment is returned as the initial function environment. The domain map we return maps each function name to the set containing the representative input value for that function.

$$\mathit{InitialFEnv}\,(pr) =$$

$$
\begin{array}{ll}
\{\ \Phi_0 & = \bot_{FEnv};\\
\quad \forall\ f_i \in F: & \\
\quad \tau_{f_i} & = \mathit{TypeOf}\,(f_i, pr);\\
\quad \langle v_{i,1}, \cdots, v_{i,n}, \sigma_i\rangle & = \mathcal{RIV}\,[\![\ \tau_{f_i}\ ]\!]\,;\\
\quad \mathcal{D}_0[f_i] & = \{\langle v_{i,1}, \cdots, v_{i,n}, \sigma_i\rangle\}; \forall\ f_i \in F\\
\text{in}\ \langle\ \Phi_0, \mathcal{D}_0\ \rangle\ \} &
\end{array}
$$

The function *ComputeCMAP* iteratively improves the approximations of the correctness map until no further information is added. It returns the most precise approximation as its result.

$$
\begin{array}{l}
\mathit{ComputeCMAP}\,(pr, \Phi, \mathcal{D}, \Psi) = \\
\quad \{\ \Psi' = \lambda f_i. \bigcup_{\langle v_1, \cdots, v_n, \sigma \in \mathcal{D}[f_i]\rangle} \mathcal{VE}\,[\![\ e_i\ ]\!]\,\bot_{Env}[v_1/x_1, \cdots, v_n/x_n]\sigma\Phi\Psi\ ;\\
\quad\ \ \Psi'' = \ \mathit{if}\ \Psi' \sqsubseteq \Psi\\
\qquad\qquad \mathit{then}\ \Psi'\\
\qquad\qquad \mathit{else}\ \mathit{ComputeCMAP}\,(pr, \Phi, \mathcal{D}, \Psi');\\
\quad\ \ \text{in}\ \Psi''\ \}
\end{array}
\tag{5.2}
$$

5.3.2 Verification of Deallocation within an Expression

This section gives a definition of algorithm $\mathcal{VE}$, which takes an expression e, an environment, a store, and a function environment, and returns the set of labels of deallocation commands in e that cannot be proven safe statically.

The following four clauses of $\mathcal{VE}$ show that $\mathcal{VE}$ returns the empty set for simple expressions and primitive expressions because none of these expressions can deallocate objects.

$$\mathcal{VE}\,[\![\ n\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \emptyset \tag{5.3}$$

$$\mathcal{VE}\,[\![\ b\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \emptyset \tag{5.4}$$

$$\mathcal{VE}\,[\![\ x\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \emptyset \tag{5.5}$$

$$\mathcal{VE}\,[\![\ +(se_1, se_2)\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \emptyset \tag{5.6}$$

Function $\mathcal{VE}$ looks in the correctness map Ψ to see if an application of procedure f is correct with respect to deallocation.

$$\mathcal{VE}\,[\![\ f(se_1, \cdots, se_n)\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \Psi[f] \tag{5.7}$$

Verification that a function has correct deallocation statements only is performed by procedure $\mathcal{VP}$, which tests the correctness of each function over all points in the function's domain.

Conditional statements have correct deallocation statements if both branches of the conditional are correct. The predicate cannot have any deallocation statements. The following clause verifies conditional expressions.

$$
\begin{array}{l}
\mathcal{VE}\,[\![\ \texttt{if}\,(se_0, e_1, e_2)\ ]\!]\,\rho\sigma\Phi\Psi\ = \\
\qquad \mathcal{VE}\,[\![\ e_1\ ]\!]\,\rho\sigma\Phi\Psi \quad \cup \quad \mathcal{VE}\,[\![\ e_2\ ]\!]\,\rho\sigma\Phi\Psi
\end{array}
\tag{5.8}
$$

The essence of the verification procedure for expressions is in the clause shown in Figure 5.2. This clause verifies the deallocation commands in `letrec` blocks. This clause must compute the

$$
\begin{array}{l}
\mathcal{VE}[\![\ \{\ Bs\text{—}Ds\ \mathtt{in}\ x\}\ ]\!]\,\rho\sigma\Phi\Psi\ = \\
\quad \{\ [\![\ x_1 = e_1; \ldots ; x_n = e_n\ ]\!] = Bs; \\
\quad\ \ [\![\ {}^{d_1}\mathtt{Dealloc}\,(y_1); \cdots ; {}^{d_k}\mathtt{Dealloc}\,(y_k)\ ]\!] = Ds; \\
\\
\quad\ \ \rho_0 \quad = \rho[\perp/x_1, \cdots, \perp/x_n]; \\
\quad\ \ \langle\, \rho', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}\prime}\,\rangle\ = \mathit{EvalBindings}_A\,(Bs, \Phi, \rho_0, \sigma, \perp_{DMAP}); \\
\\
\quad\ \ I \quad = \bigcup_{w \in FV\,(Bs)} \widehat{\mathit{Reachable}}\,(\rho[w], \sigma); \\
\quad\ \ R \quad = \widehat{\mathit{Reachable}}\,(x, \sigma'); \\
\\
\quad\ \ ds_{Bs} = \bigcup_{1 \le i \le n} \mathcal{VE}[\![\ e_i\ ]\!]\,\rho'\sigma'\Phi\Psi\ ; \\
\quad\ \ ds_{Ds} = \bigcup_{[\ {}^{d_i}\mathtt{Dealloc}(y_i)\ ]\!] \in Ds} \begin{cases} \emptyset & \text{when} \\ & \emptyset = \rho'[y_i] \cap I \\ & \wedge\ \ \emptyset = \rho'[y_i] \cap R \\ & \wedge\ \ \bigwedge_{y_j \neq y_i} \emptyset \overset{r}{=} \begin{pmatrix} \rho'[y_i] \\ \cap(\rho'[y_j] \cup \Delta^{-\prime}) \end{pmatrix} \\ \{d_i\} & \text{otherwise} \end{cases} \\
\quad\ \ \text{in } ds_{Bs} \cup ds_{Ds}\ \}
\end{array}
\tag{5.9}
$$

Figure 5.2: Clause to verify deallocation commands in `letrec` blocks

environment ρ' and store σ' of the `letrec` block bindings. It calls $\mathcal{VE}$ on each of the right-hand-side expressions with respect to ρ' and σ', collecting the results into ds_{Bs}. Then, it checks whether each deallocation statement labeled d_i in Ds is safe. If $\mathcal{VE}$ cannot prove that the deallocation statement labeled d_i is safe, it collects the d_i into the set ds_{Ds}. The result is the union of the unsafe deallocation statement labels in the body of the block and the unsafe deallocation statement labels in Ds. While computing set I, the set of object labels reachable from the surrounding context of a block expression, note that we use the incoming environment and store: ρ and σ, instead of the current environment and store: ρ' and σ'. We can use either σ or σ' here because the language is functional.

Procedure $\mathcal{VE}$ returns the empty set for tuple allocation and selection primitives, as shown below, because they cannot contain deallocation statements.

$$\mathcal{VE}[\![\ {}^{l}\mathtt{MakeTuple}\,(se_1, \cdots, se_m)\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \emptyset \tag{5.10}$$

$$\mathcal{VE}[\![\ \mathtt{Select}_i\,(se)\ ]\!]\,\rho\sigma\Phi\Psi \quad = \quad \emptyset \tag{5.11}$$

5.3.3 Verifying Some Examples

Now let us apply the above algorithm to both a correct and an incorrect example so that we may observe its operation.

A Correct Example

In this example, we apply procedure $\mathcal{VP}$ to pr, the following KID$^-$ program:

```
{ def f(x,y) =
    { t = ^l0 MakeTuple(x,y)
      result = ^k0 g(t);
     ---
      ^d1 Dealloc(t);
    in result };

  def g(t) =
    Select_1(t)

  def f_0() =
    ^k1 f(6,847)
}
```

Here are the domains of the abstracted versions of `f` and `g`.

$$\begin{aligned} Domain\,(\texttt{f}) &= N \times N \times \bot_{Store} \\ Domain\,(\texttt{g}) &= \{\emptyset, \{l_0\}\} \times \{[l_0 \rightarrow \langle_{Tuple}\ N, N\rangle], [l_0 \rightarrow \bot]\} \end{aligned}$$

where the following are the program dependent domains:

$$\begin{aligned} OL &= \{l_0\} \\ Ls &= \{\emptyset, \{l_0\}\} \end{aligned}$$

In order to verify the correctness of the deallocation commands in program pr, we must verify the deallocation commands in each procedure in the program for all input vectors in the domains of those procedures (Equation 5.1). For expository purposes, we short cut the iterative computation of the correctness map Ψ.

First, we verify the correctness of procedure `f`. There is only one input vector of interest for procedure `f`, so the value of $\Psi[\texttt{f}]$ is the result of $\mathcal{VE}$ applied to the body of `f` and this input vector.

$$\Psi[\texttt{f}] = \mathcal{VE}[\![\ e_f\ ]\!] \bot_{Env}[\underline{N}/\texttt{x}, \underline{N}/\texttt{y}] \bot_{Store} \Phi\Psi$$

```
e_f = { t = ^l0 MakeTuple(x,y)
        result = ^k0 g(t);
       ---
        ^d1 Dealloc(t);
       in result };
```

First, let us compute the value of ds_{Bs}, the labels of the incorrect deallocation commands in the body of the `letrec` block. We find:

$$\begin{aligned} ds_{Bs} &= \mathcal{VE}[\![\ {}^{l_0}\texttt{MakeTuple}(\texttt{x},\texttt{y})\ ]\!] \rho'\sigma'\Phi\Psi \\ &\quad \cup\ \mathcal{VE}[\![\ {}^{k_0}\texttt{g}(\texttt{t})\ ]\!] \rho'\sigma'\Phi\Psi \end{aligned}$$

by Equation 5.9, where e_f is the body of procedure `f` and ρ', σ', $\Delta^{-'}$, I, R and ls_t are computed by procedure $\mathcal{VE}$:

$$\begin{array}{rcl}
\rho' & = & \bot_{Env}\left[\begin{array}{l}\underline{N}/\mathtt{x},\\ \underline{N}/\mathtt{y},\\ \{l_0\}/\mathtt{t},\\ \underline{N}/\mathtt{result}\end{array}\right]\\
\sigma' & = & \bot_{Store}\,[l_0 \to \langle_{Tuple}\ \underline{N}, \underline{N}\rangle]\\
\Delta^{-'} & = & \emptyset\\
I & = & \mathit{Reachable}\,(\rho'[\mathtt{x}], \sigma) \cup \mathit{Reachable}\,(\rho'[\mathtt{y}], \sigma) = \emptyset\\
R & = & \mathit{Reachable}\,(\rho'[\mathtt{result}], \sigma') = \emptyset\\
ls_t & = & \rho'[\mathtt{t}] = \{\epsilon : l_0\}
\end{array}$$

Using these values, we can check the correctness of the two expressions from the body of the block expression. There are no incorrect deallocation commands in the `MakeTuple` expression, so $\mathcal{VE}$ returns the empty set. To apply $\mathcal{VE}$ to the application of procedure `g`, we must compute the entry $\Psi[g]$.

If we apply procedure $\mathcal{VE}$ to the body of procedure `g`, $\mathcal{VE}$ returns the empty set because there are no deallocation commands or procedure applications in the body of `g`. Consequently, the entry in Ψ for `g` contains the empty set.

$$\Psi[\mathtt{g}] = \emptyset$$

Going back to the verification of the body of procedure `f`, we find:

$$\begin{array}{rcl}
\mathcal{VE}\,[\![\ {}^{l_0}\mathtt{MakeTuple(x,y)}\ ]\!]\,\rho'\sigma'\Phi\Psi & = & \emptyset\\
\mathcal{VE}\,[\![\ {}^{k_0}\mathtt{g(t)}\ ]\!]\,\rho'\sigma'\Phi\Psi & = & \Psi[g]
\end{array}$$

by Equations 5.10 and 5.7. Therefore, the bindings of the `letrec` block in `f` contain no unsafe deallocation commands.

Now let us consider the deallocation command in the body of `f`. Using the values computed above, we see that the set of labels that may be deallocated, $\{l_0\}$, has a null intersection with both I and R, which are the sets of inherited and escaping locations. Therefore, this deallocation command satisfies the safety condition, so we can conclude that it will never lead to a run-time error.

Since both ds_{Bs} and ds_{Ds} are empty, the result of the call to $\mathcal{VE}$ on the body of `f` is the empty set, and the entry in Ψ for `f` also contains the emptyset.

$$\Psi[\mathtt{f}] = \emptyset$$

An Incorrect Example

Now let us apply the verification algorithm to a program containing an incorrect deallocation command. The program is a slight variation of the program from the previous example, in which procedure `g` returns its argument. We apply $\mathcal{VP}$ to the following program pr:

```
{ def f(x,y) =
    { t = ˡ⁰MakeTuple(x,y)
```

```
     result = ᵏ⁰g(t);
    ---
     ᵈ¹Dealloc(t);
   in result };

 def g(t) = t

 def f0() =
   ᵏ¹f(6,847)
}
```

Again, we verify the correctness of the deallocation commands in the program by verifying each of the procedure bodies over each input in the domains of the function. The procedures in program *pr* have the same domains as in the previous example.

Procedure g still contains no deallocation commands or procedure applications, so $\mathcal{VE}$ returns the empty set when applied to the body of g. Therefore, the entry in Ψ for g is the empty set.

$$\Psi[\mathtt{g}] = \emptyset$$

We proceed to verify the safety of the deallocation commands in procedure f. There is one input value in the domain of f to consider. We call $\mathcal{VE}$ on expression e_f and input value $\langle \underline{N}, \underline{N}, \bot_{Store}\rangle$.

$$\mathcal{VE}[\![\ e_f\ ]\!]\,[\underline{N}/\mathtt{x}, \underline{N}/\mathtt{y}]\bot_{Store}\Phi\Psi$$

where e_f is the body of procedure f:

$$e_f \quad = \quad \begin{array}{l} \mathtt{\{\ t\ =\ }{}^{l_0}\mathtt{MakeTuple(x,y)} \\ \quad \mathtt{result\ =\ }{}^{k_0}\mathtt{g(t);} \\ \mathtt{\ \ ---} \\ \quad {}^{d_1}\mathtt{Dealloc(t);} \\ \mathtt{\ \ in\ result\ \};} \end{array}$$

To apply $\mathcal{VE}$ to the body of procedure f, we first compute ρ' and σ':

$$\begin{array}{rcl} \rho' & = & \left[\begin{array}{l} \underline{N}/\mathtt{x}, \\ \underline{N}/\mathtt{y}, \\ \{l_0\}/\mathtt{t}, \\ \{l_0\}/\mathtt{result} \end{array} \right] \\ \sigma' & = & [l_0 \to \langle_{Tuple}\ \underline{N}, \underline{N}\rangle] \\ \Delta^{-\prime} & = & \emptyset \end{array}$$

Then we compute ds_{Bs}, the labels of the incorrect deallocates in the bindings of the letrec block.

$$\begin{array}{rcl} ds_{Bs} & = & \mathcal{VE}[\![\ {}^{l_0}\mathtt{MakeTuple(x,y)}\ ]\!]\,\rho'\sigma'\Phi\Psi \\ & & \cup\ \mathcal{VE}[\![\ {}^{k_0}\mathtt{g(t)}\ ]\!]\,\rho'\sigma'\Phi\Psi \end{array}$$

by Equation 5.9. Using these values, we can see that the deallocations in the bindings of the letrec block are correct, as in the previous example. Then we compute the other values needed:

$$\begin{array}{rcl} I & = & \mathit{Reachable}\,(\rho'[\mathtt{x}], \sigma') \cup \mathit{Reachable}\,(\rho'[\mathtt{y}], \sigma') = \emptyset \\ R & = & \mathit{Reachable}\,(\rho'[\mathtt{result}], \sigma') = \{\epsilon : l_0\} \\ ls_t & = & \rho'[\mathtt{t}] = \{l_0\} \end{array}$$

If we consider the deallocation command labeled d_1, we see that set of locations it may deallocate, $\{l_0\}$, intersects the set R of locations reachable from the result of the `letrec` block. This deallocation command violates the safety condition — it may lead to a dangling pointer error at run-time — so $\mathcal{VE}$ returns the set containing d_1 for the `letrec` block. Consequently, the entry in map Ψ for procedure `f` is $\{d_1\}$.

$$\Psi[\mathtt{f}] = \{d_1\}$$

5.4 An Algorithm for Inserting Deallocation Commands

This section describes a simple algorithm for inserting correct deallocation commands into KID$^-$ programs. This algorithm only deallocates objects that are directly named in the control region that bounds the lifetimes of the object. To be more complete, the algorithm would have to insert bindings from new variables to the nested components of dead structures in order to deallocate them. The details of this are left to the reader.

First, we look at the transformations we expect the deallocation insertion algorithm to perform. Then in the next four sections we develop the actual algorithms for inserting deallocation code.

5.4.1 Desired Results of Insertion Algorithm

Let us look at a few examples. In the following code fragment, we should be able to determine that variable x_3 can name the same objects as x_1 and x_2, but that x_1 and x_2 must be bound to different objects. Therefore, the best transformation would be to deallocate x_1 and x_2 but not x_3, as shown:

```
{ x1 = l1MakeTuple(3,4);
  x2 = l2MakeTuple(3,4);
  x3 = If p then x1 else x2;
  in 7 }
```

$\Rightarrow$

```
{ x1 = l1MakeTuple(3,4);
  x2 = l2MakeTuple(3,4);
  x3 = If p then x1 else x2;
 ---
  Dealloc(x1);
  Dealloc(x2);
  in 7 }
```

There is another correct way to transform this program. We could have inserted a deallocation command for identifier x_3 instead of the commands put in for identifiers x_1 and x_2, as follows:

```
{ x1 = l1MakeTuple(3,4);
  x2 = l2MakeTuple(3,4);
  x3 = If p then x1 else x2;
 ---
  Dealloc(x3);
  in 7 }
```

This transformation is not as good as the previous one because it only deallocates one of the tuples that are allocated when both could be deallocated. When inserting deallocation commands, we should try to find as many variables that are bound to non-overlapping sets of labels as possible, and insert deallocation commands on these variables.

It is not always possible to insert deallocation commands that deallocate all dead structures if we do not insert conditional deallocation commands. In the following example, we can insert deallocation commands for x_1 and x_2, but not x_3, because it may be bound to the same tuple as x_1.

```
{ x1 = l1MakeTuple(3,4);
  x2 = l2MakeTuple(3,4);
  x3 = If p then x1
       else l3MakeTuple(68,47);
 in 7 }
```

$\Rightarrow$

```
{ x1 = l1MakeTuple(3,4);
  x2 = l2MakeTuple(3,4);
  x3 = If p then x1
       else l3MakeTuple(68,47);
 ---
  Dealloc(x1);
  Dealloc(x2);
 in 7 }
```

However, if we insert a conditional deallocation command, then we can deallocate all of the tuples that are allocated, as shown below:

```
{ x1 = l1MakeTuple(3,4);
  x2 = l1MakeTuple(3,4);
  x3 = If p then x1
       else l3MakeTuple(68,47);
 ---
  Dealloc(x1);
  Dealloc(x2);
  if (x3 ≠ x1) then
    { ---
     Dealloc(x3) }
  else { };
 in 7 }
```

In fact, we can always take the set of all of the variables in a `letrec` block which are bound to objects whose lifetime is definitely contained in the lifetime of the block, and insert conditionals to guarantee that each distinct object to which the variables are bound at run time is deallocated exactly once.

Yet another way we can transform this example is to insert a call to `copy` on the true side of the conditional, so that the object bound to x_3 is always different from that bound to variable x_1. This may not make sense in this particular case, because it costs more to allocate an object than to perform an equality test (as we did in the previous transformation of this example). Inserting a call to `copy` makes sense if this expression is executed many times and it is much more likely to take the `else` branch than the `then` branch of the conditional. Then the amortized cost of the extra copy will be much less than the cost of the conditional before the deallocation command.

```
{ x1 = l1MakeTuple(3,4);
  x2 = l1MakeTuple(3,4);
  x3 = If p then
         k1copy(x1)
       else l3MakeTuple(68,47);
 ---
  Dealloc(x1);
  Dealloc(x2);
```

```
 Dealloc(x3);
in 7 }
```

The following example shows that we may have to insert code in order to name all of the objects that may be deallocated. We can insert a deallocation on variable x directly, but we must insert a binding to name the second component of the object named by x, which is also a structure that may be deallocated in the outer block expression.

```
{ x = { y = l1MakeTuple(3,4);
        z = l2MakeTuple(4,y);
       in z }
  in 7 }
```

$\Rightarrow$

```
{ x = { y = l1MakeTuple(3,4);
        z = l2MakeTuple(4,y);
       in z }
  w = Select2(x);
 ---
  Dealloc(x);
  Dealloc(w);
 in 7 }
```

5.4.2 The Algorithm

This section presents a simple algorithms for inserting deallocation commands in KID$^-$ programs. This algorithm only inserts commands to deallocate tuples that are named by variables in the program. It does not insert bindings to name components of tuples whose lifetimes are bounded by that of the block. It also does not insert conditionals after the barrier. Once the basic algorithm is understood it is straightforward to increase its effectiveness by having it insert code to deallocate the components of dead structures and insert conditional deallocation commands to deallocate all structures bound to identifiers that may be aliased.

The algorithm works in a greedy fashion on the set of identifiers bound in a block. It inserts deallocation commands for each identifier that is bound to a set of labels that satisfies these three conditions:

1. The lifetime of each of the labels in the set is enclosed by that of the block.
2. None of the labels are deallocated by one of the deallocation commands inserted earlier.
3. None of the labels are deallocated elsewhere in the program.

This algorithm is implemented by four procedures: $\mathcal{TP}$ and $\mathcal{TE}$, which transform programs and expressions, and $\mathcal{DS}$ and $\mathcal{DX}$ which return a list of deallocation statements for lists of bound variables and single bound variables in a given `letrec` block.

Here are the signatures of procedures used in the insertion algorithm:

$$\begin{aligned}
\mathcal{TP} &: Prog \rightarrow Prog \\
\mathcal{TE} &: Exp \rightarrow Env \rightarrow Store \rightarrow FEnv \rightarrow Exp \\
\mathcal{DS} &: X^* \rightarrow Env \rightarrow Store \rightarrow Ls \rightarrow Ls \rightarrow Ls \rightarrow DS \\
\mathcal{DX} &: V \rightarrow X \rightarrow Ls \rightarrow Ls \rightarrow Ls \rightarrow (DS \times Ls)
\end{aligned}$$

Procedure $\mathcal{TP}$ takes a program, computes the function environment for the program, and calls procedure $\mathcal{TE}$ to insert the appropriate deallocation statements in the body of each procedure in

$$
\begin{aligned}
&\mathcal{TP}[\![\ pr\ ]\!] = \\
&\qquad \{\ [\![\ \{\cdots f_i(x_1,\cdots,x_n) = e_i \cdots\}\ ]\!] = pr; \\
&\qquad\quad \mathcal{D}_0 \qquad\qquad\qquad\quad = \mathit{InitialDMAP}\,(pr); \\
&\qquad\quad \langle\ \Phi, \mathcal{D}\ \rangle \qquad\qquad\ \ = \mathit{ComputeFEnv}_A\,(pr, \bot_{FEnv}, \mathcal{D}_0); \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\quad \forall f_i \in \{f_0,\cdots,f_k\}, \\
&\qquad\quad \tau_{f_i} \qquad\qquad\qquad\quad = \mathit{TypeOf}\,(f_i, pr); \\
&\qquad\quad \langle v_{i,1},\cdots,v_{i,n},\sigma_i\rangle \ \ = \mathcal{RIV}[\![\ \tau_{f_i}\ ]\!]\ ; \\
&\qquad\quad [\![\ e_i'\ ]\!] \qquad\qquad\quad\ \ = \mathcal{TE}[\![\ e_i\ ]\!]\,\bot_{Env}[v_{i,1}/x_1,\cdots,v_{i,n}/x_n]\sigma_i\Phi\ ; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\quad \text{in } [\![\ \{\cdots f_i(x_1,\cdots,x_n) = e_i' \cdots\}\ ]\!]\ \}
\end{aligned}
\tag{5.12}
$$

Figure 5.3: Procedure to insert deallocation commands into programs

the program. Procedure $\mathcal{TE}$ takes an expression e and the most general environment, store, and function environment in which that expression executes. It returns a transformed expression e'.

The procedures $\mathcal{DS}$ and $\mathcal{DX}$ are used by procedure $\mathcal{TE}$ when translating `letrec` blocks. Procedure $\mathcal{DS}$ takes the set of variables bound by the `letrec` block, and the environment and store that are active in the bindings of the `letrec` block. In addition to the environment and store, it takes the set of inherited, escaping, and previously deallocated object labels. The inherited labels are those that are reachable from the context of the `letrec` block, the escaping labels are those reachable from the result of the `letrec` block, and the previously deallocated labels are those deallocated in the bindings of the `letrec` block. Procedure $\mathcal{DS}$ returns the set of deallocation commands for the identifiers that are determined to be safely deallocatable.

Procedure $\mathcal{DS}$ calls procedure $\mathcal{DX}$ on each bound identifier. Procedure $\mathcal{DX}$ is the procedure that actually generates a deallocation command for an identifier x when it is safe to deallocate the value of that identifier in a particular context. If procedure $\mathcal{DX}$ is applied to a variable x, and the deallocation safety condition is met for x, then $\mathcal{DX}$ returns a deallocation command for x. Procedure $\mathcal{DX}$ takes as input the binding of x, the variable x, and the sets of inherited, escaping, and previously deallocated object labels, and returns a set of deallocation commands and the set of object labels that would be deallocated by those commands.

5.4.3 Inserting Deallocation Commands in Programs

Procedure $\mathcal{TP}$, shown in Figure 5.3, takes a program pr and returns a new program with deallocation statements added to the bodies of each of the procedures in pr and the main expression of pr. Procedure $\mathcal{TP}$ calls procedure $\mathcal{TE}$ on each function body and the main expression of the program with the most general environment and store in which those expressions could be evaluated. It then reassembles the transformed expressions into a new program pr'.

5.4.4 Inserting Deallocation Commands in Expressions

The procedure $\mathcal{TE}$, which inserts deallocation commands into expressions, takes an expression, an environment, a store, and a function environment, and returns a new expression and the new set of labels that are deallocated during the execution of the new expression.

Procedure $\mathcal{TE}$ does not insert any deallocation statements in simple expressions and primitive expressions. The clauses shown below handle the processing of these expressions. The result returned from the function $\mathcal{TE}$ is a syntactic expression. These values are surrounded with syntax brackets, *i.e.*, $[\![\ x\ ]\!]$, to show that they are new program text.

$$
\begin{array}{rcll}
\mathcal{TE}[\![\ n\ ]\!]\rho\sigma\Phi & = & [\![\ n\ ]\!] & \text{where } n \text{ is a number} \\
\mathcal{TE}[\![\ b\ ]\!]\rho\sigma\Phi & = & [\![\ b\ ]\!] & \text{where } b \text{ is a boolean} \\
\mathcal{TE}[\![\ x\ ]\!]\rho\sigma\Phi & = & [\![\ x\ ]\!] & \text{where } x \text{ is a variable} \\
\mathcal{TE}[\![\ +(se_1, se_2)\ ]\!]\rho\sigma\Phi & = & [\![\ +(se_1, se_2)\ ]\!] &
\end{array}
$$

As shown below, no changes are made to function application expressions. All changes will be made to the body of the function f_i when it is transformed.

$$
\mathcal{TE}[\![\ {}^k f(se_1, \cdots, se_n)\ ]\!]\rho\sigma\Phi \quad = \quad [\![\ {}^k f(se_1, \cdots, se_n)\ ]\!]
$$

Procedure $\mathcal{TE}$ processes conditional expressions by generating a new conditional with both branches transformed, as shown below:

$$
\begin{array}{l}
\mathcal{TE}[\![\ \texttt{if}\,(se_0, e_1, e_2)\ ]\!]\rho\sigma\Phi \ = \\
\quad \{\ [\![\ e_1'\ ]\!] = \mathcal{TE}[\![\ e_1\ ]\!]\rho\sigma\Phi\,; \\
\quad\ \ [\![\ e_2'\ ]\!] = \mathcal{TE}[\![\ e_2\ ]\!]\rho\sigma\Phi\,; \\
\\
\quad\ \ \text{in}\ [\![\ \texttt{if}\,(se_0, e_1', e_2')\ ]\!]\ \}
\end{array}
$$

No changes need to be made to tuple manipulation primitives:

$$
\begin{array}{rcl}
\mathcal{TE}[\![\ {}^l\texttt{MakeTuple}\,(se_1, \cdots, se_m)\ ]\!]\rho\sigma\Phi & = & [\![\ {}^l\texttt{MakeTuple}\,(se_1, \cdots, se_m)\ ]\!] \\
\mathcal{TE}[\![\ \texttt{Select}_i\,(se)\ ]\!]\rho\sigma\Phi & = & [\![\ \texttt{Select}_i\,(se)\ ]\!]
\end{array}
$$

As in the procedure for verification, the processing of `letrec` blocks is where most of the work is done during program transformation. First, the environment, store, and set of object labels deallocated by the `let` block must be computed. Then new binding right-hand-sides must be generated by transforming the old bindings. Then the set of labels reachable from the result must be computed. A new set of deallocation statements is generated by calling procedure $\mathcal{DS}$ with the set of identifiers bound by the `letrec` block, the environment, the store, and the sets of reachable, allocated, and deallocated labels. Finally, the new right-hand-side expressions and deallocation statements are assembled into a new `letrec` block and returned. The definition of this clause is shown in Figure 5.4.

Procedure $\mathcal{DS}$ takes a list of the identifiers of a `letrec` block, the environment, and store of the body of the block, the set of labels of objects reachable from the context of the block, the set of labels of objects reachable from the result of the block, and the set of labels deallocated by the

$$
\begin{array}{l}
\mathcal{TE}[\![\ \{ \ Bs \text{—} Ds \ \mathtt{in} \ x\} \ ]\!] \rho\sigma\Phi \ = \\
\quad \{ \ [\![\ x_1 = e_1; \ldots; x_n = e_n \ ]\!] = Bs; \\
\quad\ \ [\![\ \mathtt{Dealloc}\,(y_1); \cdots; \mathtt{Dealloc}\,(y_k) \ ]\!] = Ds; \\
\\
\quad\ \ \rho_0 \qquad\quad = \rho[\perp/x_1, \cdots, \perp/x_n]; \\
\quad\ \ \langle \ \rho', \sigma', \Delta^-, \Delta^{\mathcal{D}} \ \rangle = \\
\qquad\qquad\qquad \mathit{EvalBindings}_A\,(Bs, \Phi, \rho_0, \sigma, \perp_{DMAP}); \\
\\
\quad\ \ [\![\ e'_1 \ ]\!] \quad = \mathcal{TE}[\![\ e_1 \ ]\!] \rho'\sigma'\Phi \ ; \\
\qquad\qquad\quad \vdots \\
\quad\ \ [\![\ e'_n \ ]\!] \quad = \mathcal{TE}[\![\ e_n \ ]\!] \rho'\sigma'\Phi \ ; \\
\quad\ \ [\![\ Bs' \ ]\!] = [\![\ x_1 = e'_1; \ldots; x_n = e'_n \ ]\!] \ ; \\
\\
\quad\ \ \Delta^-{}_1 \qquad = \rho'[y_1]; \\
\qquad\qquad\quad \vdots \\
\quad\ \ \Delta^-{}_k \qquad = \rho'[y_k]; \\
\quad\ \ \Delta^{-\prime} \qquad = \Delta^- \cup \bigcup_i \Delta^-{}_i; \\
\\
\quad\ \ I \qquad\quad = \bigcup_{w \in FV\,(Bs)} \widehat{\mathit{Reachable}}\,(\rho[w], \sigma); \\
\quad\ \ R \qquad\quad = \widehat{\mathit{Reachable}}\,(\rho'[x], \sigma'); \\
\\
\quad\ \ [\![\ Ds' \ ]\!] = \mathcal{DS}[\![\ x_1, \cdots, x_n \ ]\!] \rho'\sigma' I R \Delta^{-\prime} \\
\quad\ \ \text{in } [\![\ \{ \ Bs' \text{—} Ds; Ds' \ \mathtt{in} \ x\} \ ]\!] \ \}
\end{array}
$$

Figure 5.4: Clause to insert deallocation commands in `letrec` blocks

block bindings. It returns a deallocation statement for each bound identifier whose value satisfies Condition 5.1, and the set of labels deallocated by those deallocation commands. It calls procedure $\mathcal{DX}$ on each identifier. Procedure $\mathcal{DX}$ generates a deallocation command for each identifier that satisfies the safety condition.

$$
\begin{array}{l}
\mathcal{DS}[\![\ x_1, \cdots, x_n \ ]\!] \rho\sigma I R \Delta^- \ = \\
\quad \{ \ \langle \ [\![\ Ds_1 \ ]\!] , \Delta^-{}_1 \ \rangle = \mathcal{DX}(\rho'[x_1])[\![\ x_1 \ ]\!] I R \Delta^- \ ; \\
\qquad\qquad\qquad\qquad \vdots \\
\quad\ \ \langle \ [\![\ Ds_n \ ]\!] , \Delta^-{}_n \ \rangle = \mathcal{DX}(\rho'[x_n])[\![\ x_n \ ]\!] I R \left(\Delta^- \cup \left(\bigcup_{i<n} \Delta^-{}_i \right) \right) \ ; \\
\quad\ \ \text{in } [\![\ Ds_1; \cdots; Ds_n \ ]\!] \ \}
\end{array}
$$

5.4.5 Generating Deallocation Statements

Procedure $\mathcal{DX}$ takes the value of a bound variable x_i, the variable x_i, and the set of labels inherited by, escaping from, and deallocated by the surrounding `letrec` block. It returns a deallocation command for x_i if it is safe to deallocate the value of x_i upon termination of the surrounding `letrec` block. The value of x_i may be safely deallocated if x_i is bound to a reference to a structure and that structure is allocated within the current `letrec` block, is not reachable from the result of that `letrec` block, and cannot be deallocated by any other deallocation command.

$$
\begin{array}{lcl}
\mathcal{DX}(\perp)[\![\ x\ ]\!] I R \Delta^- & = & \langle\ [\![\ ]\!], \emptyset\ \rangle \\
\mathcal{DX}(\underline{N})[\![\ x\ ]\!] I R \Delta^- & = & \langle\ [\![\ ]\!], \emptyset\ \rangle \\
\mathcal{DX}(\underline{B})[\![\ x\ ]\!] I R \Delta^- & = & \langle\ [\![\ ]\!], \emptyset\ \rangle \\
\mathcal{DX}(ls)[\![\ x\ ]\!] I R \Delta^- & = & \mathit{if}\ \ \emptyset = ls \cap I \\
 & & \qquad \wedge\ \emptyset = ls \cap R \\
 & & \qquad \wedge\ \emptyset = ls \cap \Delta^- \\
 & & \quad \mathit{then}\ \langle\ [\![\ \texttt{Dealloc}(x)\ ]\!], ls\ \rangle \\
 & & \quad \mathit{else}\ \langle\ [\![\ ]\!], \emptyset\ \rangle
\end{array}
$$

The final clause of $\mathcal{DX}$ actually inserts all of the deallocation commands. The set of object labels to which x may be bound is ls, the set of locations passed into the current expression from the surrounding context is I, the set of locations reachable from the result of the expression is R, and the set of locations deallocated elsewhere is Δ^-. A deallocation is only inserted if ls is disjoint from I, if ls is disjoint from R and if ls is disjoint from Δ^-. If these three conditions are met, then a deallocation command is returned, along with the set ls of locations it may deallocate.

A more aggressive algorithm for inserting deallocation commands would examine the contents of any tuple it might deallocate to see if any of its components was also a structure that could be deallocated. If so, then more deallocation commands could be inserted, along with corresponding bindings of new variables to selection expressions in order to name the appropriate tuple components.

We do not present a more aggressive algorithm here. A more aggressive algorithm is basically the same as the one just discussed but augmented in places to track more information and to generate more complicated deallocation code. In Chapter 10, we discuss the deallocation command insertion algorithm that we implemented.

5.4.6 Transforming Some Examples

In this section we apply function $\mathcal{TP}$ to an example to see the process of inserting deallocation commands. We will walk through the transformation of the example from Section 5.3.3 with the deallocation statement removed. For reference, here is the text of the modified program pr:

```
{ def f(x,y) =
    { t = ˡ⁰MakeTuple(x,y)
      result = ᵏ⁰g(t);
    in result };

  def g(t) =
    Select₁(t)

  def f₀() =
    ᵏ¹f(6,847);
}
```

First, we determine the types of f and g in program pr:

$$\begin{aligned} \mathit{TypeOf}\,(\mathtt{f}, pr) &= (N \times N) \to N \\ \mathit{TypeOf}\,(\mathtt{g}, pr) &= \langle_{\mathit{Tuple}}\ N, N\rangle \to N \end{aligned}$$

We use these types to construct representative input vectors:

$$\begin{aligned} \mathcal{RIV}\,[\![\ (N \times N) \to N\ ]\!] &= \langle \underline{N}, \underline{N}, \bot_{\mathit{Store}} \rangle \\ \mathcal{RIV}\,[\![\ (\langle_{\mathit{Tuple}}\ N, N\rangle \to N\ ]\!] &= \langle \{l_{-1}\}, [l_{-1} \to \langle_{\mathit{Tuple}}\ \underline{N}, \underline{N}\rangle] \rangle \end{aligned}$$

Procedure $\mathcal{TP}$ (Equation 5.12) computes the function environment for the program and then constructs the following program:

$$\begin{array}{l} \{\ \mathtt{def\ f(x,y)\ =}\ e'_f \\ \quad \mathtt{def\ g(t)\ =}\ e'_g \\ \quad \mathtt{def\ f_0()\ =}\ e'_{f_0} \\ \} \end{array}$$

where

$$\begin{aligned} e'_{\mathtt{f}} &= \mathcal{TE}\,[\![\ e_{\mathtt{f}}\ ]\!]\ \bot_{\mathit{Env}}[\underline{N}/\mathtt{x}, \underline{N}/\mathtt{y}]\bot_{\mathit{Store}}\Phi \\ e'_{\mathtt{g}} &= \mathcal{TE}\,[\![\ e_{\mathtt{g}}\ ]\!]\ \bot_{\mathit{Env}}[\{l_{-1}/\mathtt{t}\}][l_{-1} \to \langle_{\mathit{Tuple}}\ \underline{N}, \underline{N}\rangle]\Phi \\ e'_{\mathtt{f}_0} &= \mathcal{TE}\,[\![\ e_{\mathtt{f}_0}\ ]\!]\ \bot_{\mathit{Env}}\bot_{\mathit{Store}}\Phi \end{aligned}$$

and $e_{\mathtt{f}}$ is the body of procedure f, and $e_{\mathtt{g}}$ is the body of procedure g, and $e_{\mathtt{f}_0}$ is the body of procedure $\mathtt{f}_0$.

Let us follow the transformation of the body of procedure f. First we need to compute a number of values:

$$\begin{aligned} [\![\ \{\mathit{BS}\ \mathtt{---}\ \mathit{DS}\ \mathtt{in}x\}\ ]\!] &= e_{\mathtt{f}} \\ [\![\ \mathtt{t} = e_1;\ \mathtt{result} = e_2\ ]\!] &= \mathit{BS} \\ [\![\ ]\!] &= \mathit{DS} \\ \rho' &= [\{l_0\}/\mathtt{t}, \underline{N}/\mathtt{result}, \underline{N}/\mathtt{x}, \underline{N}/\mathtt{y}] \\ \sigma' &= [l_0 \to \langle_{\mathit{Tuple}}\ \underline{N}, \underline{N}\rangle] \\ \Delta^{-\prime} &= \emptyset \end{aligned}$$

where ρ' and σ' are the environment and store of the body of expression $e_{\mathtt{f}}$ and $\Delta^{-\prime}$ is the set of labels of objects deallocated in $e_{\mathtt{f}}$.

These values are computed by applying *EvalBindings* to the bindings of the letrec block, the current activation label, and the current function environment. This procedure finds the fixpoint of the resulting environment, store, and set of deallocated objects' labels. From these values, we compute the additional values necessary to test the safety of deallocating the value bound to each identifier at run-time:

$$\begin{aligned} I &= \emptyset \\ R &= \emptyset \end{aligned}$$

No labels are inherited by the body of procedure `f` from the surrounding context, and no labels are returned from `f`.

Now we apply function $\mathcal{DS}$ to the bound identifiers of the `letrec` block. This procedure calls $\mathcal{DX}$ on variables `t` and `result` and the value to which that identifier is bound, along with I, R, and $\Delta^{-\prime}$:

$$\mathcal{DX}(\{l_0\})[\![\ \texttt{t}\ ]\!] IR\Delta^- \quad = \quad \langle\ [\![\ \texttt{Dealloc(t)}\ ]\!], \{\texttt{l}_0\}\ \rangle$$

This call returns a deallocation command for identifier `t` and a set containing object label l_0 because it is safe to deallocate the value bound to `t` upon termination of the control region containing the block bindings.

The call to $\mathcal{DX}$ on identifier `result` follows:

$$\mathcal{DX}(\underline{N})[\![\ \texttt{result}\ ]\!] IR\Delta^- \quad = \quad \langle\ [\![\ \ ]\!], \emptyset\ \rangle$$

No deallocation command is returned in this case because `result` is not bound to any objects.

The other two procedures in the program, `g` and $\texttt{f}_0$, are unchanged because there are no objects safe to deallocate in those procedures.

The transformed program is:

```
{ def f(x,y) =
    { t = ˡ⁰MakeTuple(x,y)
      result = ᵏ⁰g(t);
     ---
      Dealloc(t)
    in result };

  def g(t) =
    Select₁(t)

  def f₀() =
    ᵏ¹f(6,847);
}
```

as we expected.

5.5 Summary

In this chapter we developed algorithms for performing object lifetime analysis and used this lifetime information to verify or insert object deallocation commands. This analysis technique is based on an abstraction of the operational semantics of KID$^-$.

In the next few chapters, we extend the analysis framework to handle more data types and higher-order functions. We also improve the modeling of activation labels to yield more precise information about the sharing of objects.

Chapter 6

Improving the Abstract Object Labels

In this section we look at a more informative abstraction of activation labels that yields better information about the identity and lifetime of objects allocated by programs. First, we introduce a new abstraction based on regular expressions that partitions standard activation labels into equivalence classes. Next, we present the changes to the abstract interpreter definition necessary to use these activation labels. Finally, we analyze an example using these activation labels.

6.1 A Better Abstraction of Activation Labels

In Chapter 4, we saw one way to abstract activation labels so that abstract interpretation was guaranteed to terminate. However, we lost a great deal of information about the identities of objects that is very useful in the analysis of programs. In this section, we examine more precise abstractions of activation labels that yield better results in the analysis of programs.

In Chapter 3, we saw that activation labels were composed of a sequence of expression labels separated by '.', where each expression label was the label of a particular function application in the program.

$$\epsilon.k_i.\cdots.k_{i'}$$

The abstraction of activation labels should preserve some information about the standard activation labels. In fact, we would like abstract activation labels to be exactly the same as standard activation labels except in recursive invocations of functions. Figure 6.1 shows an activation tree consisting solely of non-recursive procedure calls. It is safe to do so because the set of such labels is bounded by the size of the static call graph of a program.

Figure 6.2 shows the static call-graph of three procedures, `f`, `g`, and `h`, where `g` is a recursive procedure, and the corresponding activation tree showing the structure of the activation labels of recursive calls to `g`. We would like to distinguish the activations of the initial application of `g` inside procedure `f` from the recursive applications of `g` inside procedure `g`. We can capture this by abstracting sets of standard activation labels as regular expressions. For example, the abstract activation label 1.2^+ would represent the following set of activation labels:

$$\{1.2, 1.2.2, 1.2.2.2, \ldots\}$$

Figure 6.1: Nonrecursive activation tree

Figure 6.2: Call graph of a recursive procedure

Likewise, the abstract activation label $1.2^*.3$ represents

$$\{1.3, 1.2.3, 1.2.2.3, \ldots\}$$

which are the activation labels of the calls to procedure `h`.

We can think of the program's call graph (which can be statically determined because KID$^-$ is a first order language) as a finite automaton that accepts some set of strings. These strings are the standard activation labels. Every function represents a state in the automaton, and every application primitive represents a labeled edge. Every state in the automaton is an accept state. Our improved abstract activation labels for a program are the minimal regular expressions accepted by the finite automaton derived from the program's call-graph.

The improved AL domain, shown below, consists of regular expressions that match all possible concrete activation labels. Abstract activation labels consist of an activation label paired with an expression label using ".", the disjunction of two activation labels, or the zero or more repetitions of an activation label.

$$\alpha \in AL \quad = \quad \epsilon \mid (AL.L) \mid (AL + AL) \mid (AL)^*$$

As in the standard domain of activation labels, the abstract activation label domain is a flat domain — all abstract activation labels are above the bottom element, but each one is incomparable with all of the others. The abstraction function that maps a standard activation label into an abstract activation label chooses the regular expression that accepts that particular activation label.

Abstract object labels will now consist of pairs of our new abstract activation labels and the static `MakeTuple` labels. Abstract object references are still sets of abstract object labels.

We also extend the function environment domain to map function names to mappings from products of abstract values, stores, and activation labels to pairs of an abstract value and a store.

$$\Phi \in FEnv \quad = \quad F \rightarrow ((V^n \times Store \times AL) \rightarrow (V \times Store))$$

6.2 Example Abstraction Operators for Activation Labels

The function that abstracts activation labels depends on program structure. Abstract activation labels form equivalence classes for different paths through the call graph. The domain of abstract activations corresponds to the minimal set of regular expressions that name all the paths that start at the root of the call graph and end at each node in the call graph.

We can define an abstraction function for the program whose call graph is shown in Figure 6.1. This program has four procedures, `p`, `q`, `r`, `s`, and $\mathtt{f}_0$ and five function application expressions with labels k_1, k_2, k_3, k_4 and k_5.

The function that we need in the abstract interpreter takes an abstract label and the expression label of an application expression, and returns a new abstract activation label. This function simulates a DFA where there is a state for each acyclic path to each function, and transitions are taken on the labels of application expressions. For example, the next activation label function $\mathcal{NAL}$ for the program in Figure 6.1 looks like:

$$\begin{aligned}
\mathcal{NAL}\,(\epsilon, k_1) &= k_1 \\
\mathcal{NAL}\,(k_1, k_2) &= k_1.k_2 \\
\mathcal{NAL}\,(k_1, k_3) &= k_1.k_3 \\
\mathcal{NAL}\,(k_1.k_2, k_4) &= k_1.k_2.k_4 \\
\mathcal{NAL}\,(k_1.k_2, k_5) &= k_1.k_2.k_5 \\
\mathcal{NAL}\,(k_1.k_3, k_4) &= k_1.k_3.k_4 \\
\mathcal{NAL}\,(k_1.k_3, k_5) &= k_1.k_3.k_5 \\
\mathcal{NAL}\,(\alpha, k) &= \top \quad \text{otherwise}
\end{aligned}$$

Essentially, procedure `q` was split into two states, depending on whether it had been reached by the application labeled k_2 or the one labeled k_3.

The next activation label function for the program whose call graph is shown in Figure 6.2 is more interesting, because this program contains recursive calls. We cannot split nodes to distinguish different paths through recursive calls, because this would lead to an infinite number of nodes. The function $\mathcal{NAL}$ for this graph is:

$$\mathcal{NAL}\,(\epsilon, k_1) \quad = \quad k_1$$

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^k f(se_1, \cdots, se_n) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\quad \{ \ v_1 \quad = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \ ; \\
\qquad \vdots \\
\quad\ \ v_n \quad = \mathcal{SE}_A [\![\ se_n \ ]\!] \rho \ ; \\
\quad\ \ \alpha' \quad = \mathcal{NAL}\,(\alpha, k); \\
\quad\ \ \langle \ v, \ \sigma', \ \Delta^- \ \rangle = \Phi[f][v_1, \cdots, v_n, \sigma, \alpha']; \\
\quad\ \ \text{in} \ \langle \ v, \sigma', \Delta^-, \Delta^{\mathcal{D}} \ \rangle \ \}
\end{array}
$$

Figure 6.3: Evaluation of procedure calls with improved abstract activation labels

$$
\begin{array}{rcl}
\mathcal{NAL}\,(k_1, k_2) & = & k_1.k_2^* \\
\mathcal{NAL}\,(k_1.k_2^*, k_2) & = & k_1.k_2^* \\
\mathcal{NAL}\,(k_1.k_2^*, k_3) & = & k_1.k_2^*.k_3 \\
\mathcal{NAL}\,(\alpha, k) & = & \top \quad \text{otherwise}
\end{array}
$$

In this example, all activations of procedure g have activation label $k_1.k_2^*$. If there were more than one non-recursive path to invoke procedure g, then these would have distinct activation labels.

6.3 Extensions to Abstract Interpreter

This section describes the way to revise the abstract interpreter to compute improved activation labels. The expression evaluator now takes an abstract activation label in addition to the environment, store, and function environment that it took before.

The new evaluation rule for function applications is shown in Figure 6.3. Note that the function $\mathcal{NAL}$ is used to create a new abstract activation label given the current activation label α and the expression label k. We look in the function environment Φ to find the value of the body of the function evaluated with activation label α', which is the abstraction of the current activation label concatenated with k.

The revised abstract interpreter clause that evaluates the `MakeTuple` primitive is shown in Figure 6.4. This clause constructs a new object label from the current abstract activation label α and the expression label l. Other than that it is the same as the original abstract interpreter clause for `MakeTuple`. All other clauses of the abstract interpreter remain the same, except that activation labels are passed to the expression evaluator.

6.4 Evaluation of Examples Using Improved Activation Labels

Figure 6.5 contains an example that we saw earlier where sharing is falsely detected by the abstract interpreter using completely static object labels. Let us reexamine this example using our improved abstraction of activation labels and object labels.

Let us examine the input-output behavior of procedure g first. If g is applied to a number in context $\langle \ \rho, \ \sigma, \ \alpha \ \rangle$, it returns a reference to an object in location $\alpha : l_0$ and a store σ' which is derived from

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^{l}\texttt{MakeTuple}(se_1, \cdots, se_m) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\quad \{ \begin{array}{ll}
v_1 & = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \, ; \\
 & \vdots \\
v_m & = \mathcal{SE}_A [\![\ se_m \ ]\!] \rho \, ; \\
v_{tuple} & = \langle_{Tuple} \ v_1, \cdots, v_m \rangle \, ; \\
ol & = \alpha : l; \\
v'_{tuple} & = \sigma[ol]; \\
\sigma' & = \sigma[ol \rightarrow (v_{tuple} \sqcup v'_{tuple})]; \\
\multicolumn{2}{l}{\text{in } \langle \ \{ol\}, \sigma', \emptyset, \perp_{DMAP} \rangle \ \}}
\end{array}
\end{array}
$$

Figure 6.4: Evaluation of tuple allocation with improved abstract activation labels

```
{ def f(w) =
    { t1 = ^k0 g(w);
      w2 = w * 2;
      t2 = ^k1 g(w2);
      r = (w * w2);
      t3 = ^l3 MakeTuple(t1,t2,r);
     in t3 };
  def g(x) =
    { y = (x-21);
      t = ^l0 MakeTuple(x,y) ;
    in t }
  def f0() =
    ^k2 f(68);
};
```

Figure 6.5: Example with false sharing

store σ as follows:

$$
\sigma' \quad = \quad \sigma[\alpha : l_0 \rightarrow (\sigma[\alpha : l_0] \sqcup \langle_{Tuple} \ \underline{N}, \underline{N} \rangle)]
$$

Now, let us study the internal behavior of function `f` when applied to a number and the empty store in activation ϵ. We evaluate the bindings of the `letrec` block in the body of `f` to yield the environment ρ' and store σ':

$$
\rho' \quad = \quad \perp_{Env} \left[\begin{array}{lll}
w & \rightarrow & \underline{N} \\
t1 & \rightarrow & \{\epsilon.k_0 : l_0\} \\
w2 & \rightarrow & \underline{N} \\
t2 & \rightarrow & \{\epsilon.k_1 : l_0\} \\
r & \rightarrow & \underline{N} \\
 & & \\
t3 & \rightarrow & \{\epsilon : l_3\}
\end{array} \right]
$$

$$\sigma' \quad = \quad \perp_{Store} \left[\begin{array}{l} \epsilon.k_0 : l_0 \rightarrow \langle_{Tuple} \; \underline{N}, \underline{N} \rangle \\ \epsilon.k_1 : l_0 \rightarrow \langle_{Tuple} \; \underline{N}, \underline{N} \rangle \\ \epsilon : l_3 \rightarrow \langle_{Tuple} \; \underline{N}, \underline{N}, \underline{N} \rangle \end{array} \right]$$

Using the more precise abstraction of activation labels, we can distinguish between the tuples to which `t1` and `t2` are bound. We can tell that they must be different objects. Therefore, in the body of procedure $\texttt{f}_0$, we can insert code to deallocate all three tuples allocated, rather than only two. This degree of precision can be very useful.

6.5 Summary

There are many ways in which we can abstract activation labels. In this chapter we discussed one way to abstract activation labels that improves the effectiveness of the analysis compared to the abstracted activation labels that we used in Chapter 4. We use this abstraction for the remainder of the thesis, and we use a variation of these abstract activation labels in our implementation of the lifetime analysis.

Chapter 7

Abstracting and Analyzing Arrays

The abstract interpretation of arrays is different from that of tuples: the size of an array is computed at run-time, while the size of a tuple is fixed at compile-time. Section 7.1 discusses our approach to abstracting arrays — we summarize all elements of an array by one abstract value.

This array abstraction leads to problems determining whether there is sharing among the elements of the arrays. Section 7.2 discusses an improved array abstraction that contains an annotation informing whether any elements in the array are shared.

Sometimes it is difficult to define an array using `MakeArray`, even though the program fits nicely in the single-assignment paradigm. Id has I-structure arrays to extend the single-assignment paradigm beyond the functional subset. I-structures are non-functional, single-assignment arrays whose presence greatly increase the expressiveness of the language, and only slightly increase the complexity of lifetime analysis. For example, writing a function that finds the inverse of a permutation takes $O(n^2)$ space and time when written using `MakeArray`, but can easily be written in $O(n)$ time and space using I-structures. Section 7.3 discusses the addition of I-structures to our instrumented and abstract interpreters and their impact on the deallocation safety condition.

7.1 Abstract Interpretation of Arrays

Arrays are aggregate objects whose size is not determined until run-time. In the interpreter, the objects must be represented by structures with a fixed number of components because we require abstract interpretation to take a finite amount of time. We summarize the value of an array of arbitrary size as an abstract array with a single element. Our array abstraction has a single component that represents all of the elements of the concrete array. This single abstract element is the least upper bound of the abstraction of all of the concrete array elements. We call this summarization *spatial summarization*. Spatial summarization combines information about an uncertain reference or spatial path, not about an uncertain control path.

For example, consider the following concrete array of tuples:

$$\langle_{Array}\ 3, l_1, l_2, l_3\rangle$$

where 3 denotes the length of the array and l_1, l_2 and l_3 are the labels of concrete tuples. The abstraction of this array would be an abstract array with one element summarizing all of the

$$Abs_{Array}\left(\langle_{Array}\ n, v_1, \cdots, v_n\rangle\right) \quad = \quad \left\langle_{Array} \bigsqcup_i v_i \right\rangle$$

Figure 7.1: Array abstraction operator

$$\langle_{Array}\ v_0\rangle \sqcup_{Array} \langle_{Array}\ v_1\rangle \quad = \quad \langle_{Array}\ v_0 \sqcup_V v_1\rangle$$

Figure 7.2: Array least upper bound operator

$$\langle_{Array}\ v_0\rangle \sqsubseteq_{Array} \langle_{Array}\ v_1\rangle \quad = \quad v_0 \sqsubseteq_V v_1$$

Figure 7.3: Array ordering operator

---

elements of the standard array:

$$\left\langle_{Array}\ \{\hat{l_1}, \hat{l_2}, \hat{l_3}\}\right\rangle .$$

The element $\{\hat{l_1}, \hat{l_2}, \hat{l_3}\}$ indicates that the components of the standard array could be any one of the abstract tuples named by $\hat{l_1}$, $\hat{l_2}$ or $\hat{l_3}$.

If we had subscript range information, we might be able to abstract the elements of an array into a small number of elements that represent the values that could be present in subregions of the array under the standard interpretation. In that case, an array would be represented as a set of intervals and the abstract values that summarize the components of the standard array contained in those intervals. The use of range information during abstract interpretation is an area for further research.

7.1.1 The Abstract Array Domain

We add the following definition of arrays to our abstract domains, and revise the definition of the abstract store and storable value domains.

$$\begin{array}{lcll} v_{array} \in \mathit{Array} & = & \langle_{Array}\ V\rangle & \text{Arrays} \\ sv \in SV & = & \mathit{Tuple} + \mathit{Array} & \text{Storable Values} \\ \sigma \in \mathit{Store} & = & L \rightarrow SV & \text{Stores} \end{array}$$

Figure 7.1 contains the function Abs_{Array}, which maps standard array values into abstract array values. Figure 7.2 contains the least upper bound operator on abstract arrays, and Figure 7.3 contains the ordering operator for abstract arrays.

These domains, along with the added ordering and abstraction operators, allow us to revise the abstract interpreter to model arrays.

7.1.2 Abstracting the Array Primitives

The following two clauses, 7.1 and 7.2, give the abstracted evaluation rules for the array primitives. As in the standard interpreter, the `MakeArray` primitive is subscripted with the name of a function

f_i and takes a length value n and r values to be passed to the calls to f_i. The length n is ignored, because abstract arrays all contain a single component. The abstract interpretation of this primitive uses the static label l of the primitive directly as the object label of the abstract array created. In this way, it resembles the abstract interpretation of the `MakeTuple` primitive.

First, we compute the value of the application of function f_i to the input value consisting of the values $\underline{N}$, the r input values, and the current store. As with the interpretation of function applications, we look up the returns value of the function application in the function environment and add the input value to the interesting domain of function f_i in the new domain map delta $\Delta^{\mathcal{D}}$. We use the result value of the function application as the representative element value of the array.

$$\begin{array}{ll} \mathcal{E}_A [\![\ {}^{l}\texttt{MakeArray}_{f_i}(se_0, se_1, \cdots, se_r) \ ]\!] \rho\sigma\alpha\Phi \ = & \\ \quad \{ \ v_1 & = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \ ; \\ & \vdots \\ \quad v_r & = \mathcal{SE}_A [\![\ se_r \ ]\!] \rho \ ; \\ \quad \alpha' & = \mathcal{NAL}\,(\alpha, l); \\ \quad \langle \ u, \ \sigma', \ \Delta^- \ \rangle & = \Phi[f_i][\underline{N}, v_1, \cdots, v_r, \sigma, \alpha']; \\ \quad v_{array} & = \langle_{Array} \ u \rangle \ ; \\ \quad ol & = \alpha : l; \\ \quad v'_{array} & = \sigma'[ol]; \\ \quad \sigma'' & = \sigma'[ol \to (v_{array} \sqcup v'_{array})]; \\ \quad \Delta^{\mathcal{D}}[f_i] & = \{ \langle \underline{N}, v_1, \cdots, v_r, \sigma, \alpha' \rangle \}; \\ \quad \text{in } \langle \ \{ol\}, \sigma'', \Delta^-, \Delta^{\mathcal{D}} \ \rangle \ \} & \end{array} \tag{7.1}$$

The abstract interpretation of the `Fetch` primitive is very similar to the abstraction of the $\texttt{Select}_i$ primitive. `Fetch` takes two values: a set ls of labels and an index. `Fetch` takes the least upper bound of the arrays to which each of the labels in ls is bound in store σ, and then returns the element value of that array.

$$\begin{array}{ll} \mathcal{E}_A [\![\ {}^{k}\texttt{Fetch}(se_1, se_2) \ ]\!] \rho\alpha\sigma\Phi \ = & \\ \quad \{ \ ls & = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \ ; \\ \quad \langle_{Array} \ v \rangle & = \bigsqcup_{ol \in ls} \sigma[ol]; \\ \quad \text{in } \langle \ v, \sigma, \emptyset, \bot_{DMAP} \ \rangle \ \} & \end{array} \tag{7.2}$$

The abstraction of the `Bounds` primitive is very simple. It ignores its argument and returns an abstract integer as its result.

$$\begin{array}{l} \mathcal{E}_A [\![\ {}^{k}\texttt{Bounds}(se) \ ]\!] \rho\alpha\sigma\Phi \ = \\ \quad \{ \text{ in } \langle \ \underline{N}, \sigma, \emptyset, \bot_{DMAP} \ \rangle \ \} \end{array}$$

Now that we have an abstraction of array values and have augmented the abstract interpreter with clauses for the array primitives, let us examine some array program examples and see how our lifetime analysis algorithm performs.

7.1.3 Example Array Programs

The first example we look at is shown below. It consists of a function `f1` takes three numbers and constructs an array containing a different tuple in each element. The function `g1` is the function that defines each of the array elements.

```
def g1 (i, x, y) =
  ˡ⁰MakeTuple(x,y,i);

def f1 (n, x, y) =
  ˡ¹MakeArray_g1(n, x, y);
```

Our abstract interpretation of this example computes the following representation for the value of ρ and σ within f_1's body:

$$\begin{aligned} \rho[a] &= \{l_1\} \\ \sigma[l_1] &= \langle_{Array}\ \{l_0\}\rangle \\ \sigma[l_0] &= \langle_{Tuple}\ \underline{N}, \underline{N}, \underline{N}\rangle \end{aligned}$$

That is, variable `a` is bound to an array labeled l_1 which contains a a three-tuple of numbers labeled l_0 as its element.

We can determine that the lifetime of the array labeled l_1 is bounded by the lifetime of `f1`, because l_1 is not reachable from the labels inherited by `f1` (the empty set) and because l_1 is not returned as part of the result of `f1` — a single number is returned. The same is true of the tuple labeled l_0 — its lifetime is bounded by that of procedure `f1`.

Now consider the following example, which is similar to the previous one, except that the tuple labeled l_0 is allocated by procedure `f2` and passed to the procedure `g2` that computes the elements of the array l_1. Thus, `f2` allocates an array where a tuple is shared by each of the elements.

```
def f2 (n, x, y) =
  { t = l0MakeTuple(x,y,4);
    a = l1MakeArrayg2(n, t);
  in 3 };

def g2 (i, t) = t;
```

The representation computed for the value of `a` is the same as in the first example:

$$\begin{array}{rcl} a & = & \{l_1\} \\ \sigma[l_1] & = & \langle_{Array}\ \{l_0\}\rangle \\ \sigma[l_0] & = & \langle_{Tuple}\ \underline{N}, \underline{N}, \underline{N}\rangle \end{array}$$

The variable `a` is bound to an array labeled l_1 containing a tuple or tuples labeled l_0. In this example, we can determine that the lifetimes of array l_1 and tuple l_0 are bounded by the lifetime of procedure `f2`.

In both of these examples, we can verify that it is safe to deallocate the array bound to `a` when either `f1` or `f2` terminate, because label l_1 is allocated within the body of `f1` and `f2`, l_1 does not escape, and l_1 cannot be deallocated elsewhere.

There is one fact we have not been able to uncover using our lifetime analysis, and that is that in the first example, each element of the concrete array is distinct, and that in the second example, each element of the concrete array is the same. If there is no sharing, then the compiler may insert code to deallocate each element of the array. If there is sharing, then deallocation of the elements becomes a little more difficult because we cannot deallocate any element more than once. We can work around this problem with run-time support. The run-time code that deallocates the elements of an array must keep track of the objects it has deallocated to ensure that it deallocates each unique element of the array exactly once.

The first example actually has no sharing. But because the abstraction does not yield sharing information, the compiler must generate code that carefully deallocates each distinct element of the array `a` for both `f1` and `f2`. This strategy of code generation is safe, but is less efficient than if we could determine that there was no sharing of elements in procedure `f1`.

7.2 Sharing Analysis in Arrays

In the previous section, we defined an abstraction of arrays and showed how to perform lifetime analysis on programs containing arrays. We also saw that the analysis does not capture an important fact about the arrays, namely, whether the elements of the array are shared are not.

This section investigates a change to the representation of abstract arrays in the abstract interpreter so that the analysis yields sharing information. The approach we take enables us to determine whether two elements of an array are completely distinct or whether they may be shared at some level.

$$
\begin{array}{rcl}
\langle_{Array}\ s_0, v_0\rangle \sqcup_{Array} \langle_{Array}\ s_1, v_1\rangle & = & \langle_{Array}\ (s_0 \sqcup_S s_1), (v_0 \sqcup_V c_1)\rangle \\
\mathit{UnShared} \sqcup_S \mathit{UnShared} & = & \mathit{UnShared} \\
\mathit{UnShared} \sqcup_S \mathit{Shared} & = & \mathit{Shared} \\
\mathit{Shared} \sqcup_S \mathit{UnShared} & = & \mathit{Shared} \\
\mathit{Shared} \sqcup_S \mathit{Shared} & = & \mathit{Shared}
\end{array}
$$

Figure 7.4: Improved array least upper bound operators

7.2.1 Modeling Sharing in the Abstract Array Domain

In order to track the sharing of array elements, we add an annotation to each abstract array that indicates whether the array components may be shared or not. This sharing annotation is drawn from domain S, which consists of two values: *Shared* and *Unshared*, where $\mathit{Unshared} \sqsubseteq \mathit{Shared}$.

$$
\begin{array}{lcll}
s \in S & = & \mathit{Shared} + \mathit{Unshared} & \text{Sharing Predicate} \\
v_{array} \in \mathit{Array} & = & \langle_{Array}\ S, V\rangle & \text{Arrays} \\
sv \in SV & = & \mathit{Tuple} + \mathit{Array} & \text{Storable Values} \\
\sigma \in \mathit{Store} & = & L \rightarrow SV & \text{Stores}
\end{array}
$$

If we have an array of nested structures, we take *Unshared* to mean that the structures stored in each element of the array are completely unaliased from the structures stored in every other element of the array.

Figure 7.4 contains the least upper bound operator on abstract arrays, Figure 7.5 contains the ordering operator for abstract arrays and Figure 7.6 contains the abstraction operator for arrays with sharing.

7.2.2 Abstracting the Array Primitives with Sharing

The clauses of the interpreter must be augmented to compute the proper sharing information. The only change is to `MakeArray`, which generates either a shared array or an unshared array. A call to $\mathsf{MakeArray}_{f_i}$ generates an unshared array only if all of the labels reachable from the the application of procedure f_i are disjoint from the locations reachable from the arguments to the call to $\mathsf{MakeArray}$. Since none of the inherited labels are reachable from the element value resulting from the application of f_i, all of the labels reachable from the element value must be allocated within the application, and none may be shared among elements of the array.

$$
\mathcal{E}_A [\![\ {}^{k}\mathtt{MakeArray}_{f_i}\,(se_0, se_1, \cdots, se_r)\ ]\!] \rho\sigma\alpha\Phi \ =
$$

$$
\begin{array}{rcl}
\langle_{Array}\ s_0, v_0\rangle \sqsubseteq_{Array} \langle_{Array}\ s_1, v_1\rangle & = & (s_0 \sqsubseteq_S s_1) \wedge (v_0 \sqsubseteq_V v_1) \\
\mathit{UnShared} \sqsubseteq_S \mathit{UnShared} & = & \mathsf{True} \\
\mathit{UnShared} \sqsubseteq_S \mathit{Shared} & = & \mathsf{True} \\
\mathit{Shared} \sqsubseteq_S \mathit{UnShared} & = & \mathsf{False} \\
\mathit{Shared} \sqsubseteq_S \mathit{Shared} & = & \mathsf{True}
\end{array}
$$

Figure 7.5: Improved array ordering operators

$$
\begin{array}{rcl}
\mathit{Abs}_{Array}\left(\langle_{Array}\ n, v_1, \cdots, v_n\rangle\right) & = & \{\ s =\ \mathit{if}\left(\bigwedge_{1\le i,j\le n} v_i \neq v_j\right) \\
& & \qquad \mathit{then\ Unshared} \\
& & \qquad \mathit{else\ Shared}; \\
& & \quad v = \bigsqcup_{1\le i\le n} v_i; \\
& & \quad \text{in } \langle_{Array}\ s, v\rangle\ \}
\end{array}
$$

Figure 7.6: Abstraction operator for arrays with sharing

$$
\begin{array}{ll}
\{\ v_1 & = \mathcal{SE}_A[\![\ se_1\ ]\!]\rho\ ; \\
 & \ \vdots \\
\ \ v_r & = \mathcal{SE}_A[\![\ se_r\ ]\!]\rho\ ; \\
\ \ \alpha' & = \mathcal{NAL}\,(\alpha, l); \\
\ \ \langle\ u, \sigma', \Delta^-\ \rangle & = \Phi[f_i][\underline{N}, v_1, \cdots, v_r, \sigma, \alpha']; \\
\ \ I & = \bigcup_{v_i} \widehat{\mathit{Reachable}}\,(v_i, \sigma); \\
\ \ R & = \widehat{\mathit{Reachable}}\,(u)\sigma'; \\
\ \ v_{array} & = \ \mathit{if}\ R \cap I = \emptyset \\
 & \qquad \mathit{then}\ \langle_{Array}\ \mathit{Unshared}, u\rangle \\
 & \qquad \mathit{else}\ \langle_{Array}\ \mathit{Shared}, u\rangle\ ; \\
\ \ ol & = \alpha : k; \\
\ \ v'_{array} & = \sigma'[ol]; \\
\ \ \sigma'' & = \sigma'[ol \rightarrow (v_{array} \sqcup v'_{array})]; \\
\ \ \Delta^{\mathcal{D}}[f_i] & = \{\langle \underline{N}, v_1, \cdots, v_r, \sigma, \alpha'\rangle\}; \\
\ \ \text{in } \langle\ \{ol\}, \sigma'', \Delta^-, \Delta^{\mathcal{D}}\ \rangle\ \} &
\end{array}
$$

The only change to the `Fetch` evaluation clause is to make it fetch components from abstract arrays annotated with sharing information.

$$
\begin{array}{l}
\mathcal{E}_A[\![\ {}^k\mathtt{Fetch}\,(se_1, se_2)\ ]\!]\rho\alpha\sigma\Phi\ = \\
\qquad \begin{array}{ll}
\{\ ls & = \mathcal{SE}_A[\![\ se_1\ ]\!]\rho\ ; \\
\ \ \langle_{Array}\ s, v\rangle & = \bigsqcup_{ol\in ls} \sigma[ol]; \\
\ \ \text{in } \langle\ v, \sigma, \emptyset, \perp_{DMAP}\ \rangle\ \} &
\end{array}
\end{array}
$$

No change is needed in the clause for the `Bounds` primitive.

7.2.3 Reexamining the Array Examples

The first example we considered in Section 7.1.3 had no sharing in it. The example is shown below for reference.

```
def g1 (i, x, y) =
  ^l0 MakeTuple(x,y,i);

def f1 (n, x, y) =
  { a = ^l1 MakeArray_g1(n, x, y);
  in 3 };
```

Our new abstract interpretation should discover that there can be no sharing in this example. The arguments to the call to `MakeArray` are:

$$\begin{array}{rcl} n & = & \underline{N} \\ v_x & = & \underline{N} \\ v_y & = & \underline{N} \end{array}$$

assuming that variables `x` and `y` are bound to integers. Thus, there are no locations reachable from the arguments to the call to MakeArray.

$$I = \emptyset$$

The result of the call to `g1` is a reference to a tuple in location l_0, so the set of reachable locations is $\{l_0\}$:

$$R = \{l_0\}$$

and the intersection of the inherited locations and the reachable locations is the empty set. Therefore, there can be no sharing between elements of the array. There is no way, without side effects, that different elements of the array could end up sharing the same location.

We end up computing the following representation for the value of `a`:

$$\begin{array}{rcl} \rho[a] & = & \{l_1\} \\ \sigma[l_1] & = & \langle_{Array}\ \mathit{Unshared}, \{l_0\}\rangle \\ \sigma[l_0] & = & \langle_{Tuple}\ \underline{N}, \underline{N}, \underline{N}\rangle \end{array}$$

That is, `a` is bound to an array labeled l_1, containing a tuple or tuples labeled l_0 as its elements, and the tuples in different elements of the array are guaranteed to be distinct.

If we determine that the above array and its components are dead in some context, then we can insert code that deallocates the array and all of its components without having to insert any run-time code to detect sharing among the components.

The second example from Section 7.1.3, which created an array containing shared elements, follows:

```
def f2 (n, x, y) =
  { t = ^l0 MakeTuple(x,y,4);
    a = ^l1 MakeArray_g2(n, t);
  in 3 };

def g2 (i, t) = t;
```

In this case, the arguments to the `MakeArray` are:

$$\begin{aligned}\rho[n] &= \underline{N} \\ \rho[v_t] &= \{l_0\}\end{aligned}$$

Therefore, the inherited locations are the set $\{l_0\}$:

$$I = \{l_0\}$$

The application of procedure `g1` yields a reference to location l_0, so the set of reachable locations R is:

$$R = \{l_0\}$$

Since the intersection of I and R is non-empty, the array representation is shared.

$$\begin{aligned}a &= \{l_1\} \\ \sigma[l_1] &= \langle_{Array}\ Shared, \{l_0\}\rangle \\ \sigma[l_0] &= \langle_{Tuple}\ \underline{N}, \underline{N}, \underline{N}\rangle\end{aligned}$$

The variable `a` is bound to an array labeled l_1 containing a tuple or tuples labeled l_0, where there is some sharing among the elements of the array.

In this example, we can still insert code to deallocate the array and its components if we determine that they are dead in some context, but we have to insert run-time code to detect sharing.

7.3 Modeling I-Structures

This section extends the instrumented and abstracted interpreters with I-structure array data types primitives. Although Id has both I-structure algebraic types and arrays, we discuss only I-structure arrays — the implementation of other I-structure types follows directly from the model of I-structure arrays. We use the array value domain, but add two new array operators: `MakeIArray` and `Store` to KID$^-$. The first subsection presents the standard semantics of I-structure arrays, and the second subsection presents the abstract semantics extended with I-structure array operators.

In KID$^-$, I-structures are created using the primitive `MakeIArray` with all slots empty, or bound to $\perp$. Elements of the array may be filled in using the primitive `Store` and dereferenced using the primitive `Fetch`. It is an error to store more than one value into a a single I-structure array slot, although we do not check this in the interpreter.

7.3.1 I-Structures in the Instrumented Interpreter

This section defines the interpreter clauses for the primitives `MakeIArray` and `Store`. The primitive `MakeIArray` takes one value: a simple expression that evaluates to length n. It returns a newly allocated array of length n where each component is initially unbound.

$$\mathcal{E}_I [\![\ {}^k\texttt{MakeIArray}(se) \ ]\!] \rho\alpha\sigma \ =$$

$$
\begin{aligned}
\{\ & ol && = \alpha : k; \\
& n && = \mathcal{SE}[\![\ se\ ]\!]\rho\ ; \\
& v_0 && = \bot; \\
& && \ \vdots \\
& v_{n-1} && = \bot; \\
& \sigma' && = \sigma[ol \rightarrow \langle_{Array}\ n, v_0, \cdots, v_{n-1}\rangle]; \\
& \text{in}\ \langle\ ol, \sigma', \{\langle\ \alpha,\ ol\ \rangle\}, \emptyset, \emptyset\ \rangle\ \}
\end{aligned}
$$

The primitive `Store` takes three values: a reference ol to an I-structure array a, an index i, and a value v, and returns the reference to the array and a store that has the ith component of a bound to v. The interpreter records the fact that a side-effect was done on the object labeled ol by returning a reference event for ol in activation α.

$$
\begin{aligned}
&\mathcal{E}_I[\![\ {}^k\texttt{Store}(se_1, se_2, se_3)\ ]\!]\rho\alpha\sigma\ = \\
&\qquad \{\ ol = \mathcal{SE}[\![\ se_1\ ]\!]\rho\ ; \\
&\qquad\ \ i\ \ = \mathcal{SE}[\![\ se_2\ ]\!]\rho\ ; \\
&\qquad\ \ v_i' = \mathcal{SE}[\![\ se_3\ ]\!]\rho\ ; \\
&\qquad\ \ \langle_{Array}\ n, v_0, \cdots, v_i, \cdots, v_{n-1}\rangle = \sigma[ol]; \\
&\qquad\ \ \sigma' = \sigma[ol \rightarrow \langle_{Array}\ n, v_0, \cdots, v_i \sqcup v_i', \cdots, v_{n-1}\rangle]; \\
&\qquad\ \ \text{in}\ \langle\ ol, \sigma', \emptyset, \emptyset, \{\langle\ \alpha,\ ol\ \rangle\}\ \rangle\ \}
\end{aligned}
$$

Now that we have seen the definition of the instrumented interpreter clauses for handling I-structures, we can go on the definition of the abstracted I-structure domains and the abstracted I-structure interpreter clauses.

7.3.2 I-Structures in the Abstract Interpreter

In the abstracted interpreter, we use the array domains with sharing information. Whenever we store into an I-structure array, we make that array be a shared array.

The following two clauses give the abstracted evaluation rules for I-structure array data structure primitives. The primitive MakeIArray constructs an abstract array with no sharing whose components are undefined. The primitive `Store` updates the component of the array to the least upper bound of the current array element and the new value. Storing into an I-structure array may potentially introduce sharing, so we upgrade the sharing indicator of the array to *Shared* when a `Store` is performed.

$$
\begin{aligned}
&\mathcal{E}_A[\![\ {}^k\texttt{MakeIArray}(se_1)\ ]\!]\rho\sigma\alpha\Phi\ = \\
&\qquad \{\ v_{array} = \langle_{Array}\ \mathit{Unshared}, \bot\rangle\ ; \\
&\qquad\ \ ol \qquad\ = \alpha : l; \\
&\qquad\ \ v_{array}' = \sigma[ol]; \\
&\qquad\ \ \sigma' \qquad\ = \sigma[ol \rightarrow (v_{array} \sqcup v_{array}')]; \\
&\qquad\ \ \text{in}\ \langle\ \{ol\}, \sigma', \emptyset, \bot_{DMAP}\ \rangle\ \} \\
&\mathcal{E}_A[\![\ {}^k\texttt{Store}(se_1, se_2, se_3)\ ]\!]\rho\alpha\sigma\Phi\ = \\
&\qquad \{\ ls = \mathcal{SE}_A[\![\ se_1\ ]\!]\rho\ ; \\
&\qquad\ \ v\ \ = \mathcal{SE}_A[\![\ se_3\ ]\!]\rho\ ; \\
&\qquad\ \ \sigma' = \lambda ol. \begin{cases} \sigma[ol] \sqcup \langle_{Array}\ \mathit{Shared}, v\rangle & \text{if } ol \in ls \\ \sigma[ol] & \text{otherwise} \end{cases} \\
&\qquad\ \ \text{in}\ \langle\ ls, \sigma', \emptyset, \bot_{DMAP}\ \rangle\ \}
\end{aligned}
$$

7.3.3 Effect Of I-Structures on Deallocation Safety Conditions

The introduction of I-structures into KID$^-$ has introduced a new mechanism by which objects can escape from a given activation. Previously, objects could be passed into an activation through the environment as inherited values or arguments, or they could be passed out of the activation as part of the results. Now, an I-structure with empty slots can be passed into an activation, and objects allocated within the activation can be stored into the empty slots and escape from the activation. Thus it is now possible for objects allocated within an activation to escape via the inherited objects.

However, this new path for escaping objects does not significantly change the criteria that we use to decide that a particular activation contains an object's lifetime. We now must determine that an object is not reachable from the result of an expression or from the objects inherited from the surrounding environment *after the expression has executed*. The only change in the tests is which store is used to determine reachability from inherited objects. Previously, we used the incoming store to determine reachability; now we must use outgoing store to determine reachability.

Let us again consider the canonical `letrec` block with deallocation commands that we would like to verify:

$$
\begin{array}{rl}
e = & \{\ x_1 = e_1; \\
 & \quad \vdots \\
 & x_n = e_n; \\
 & — \\
 & \texttt{Dealloc}\,(y_1) \\
 & \quad \vdots \\
 & \texttt{Dealloc}\,(y_m) \\
 & \text{in } x_j\ \}
\end{array}
$$

where the environment, store, and function environment in which e is to be evaluated are ρ, σ, and Φ, respectively.

We compute environment ρ' and store σ', the resulting environment and store for the block bindings, Δ^-, the set of labels deallocated by the block bindings, and v, which is the result of the evaluation of the expression, as shown below. In addition, we compute R, the set of object labels reachable from the result of the expression, and I, the set of object labels reachable from the free variables of the expression.

$$
\begin{array}{rcl}
\rho_0 & = & \rho[\perp/x_1, \cdots, \perp/x_n] \\
\langle\ \rho', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}}\ \rangle & = & \mathit{EvalBindings}_A\,(\llbracket\ x_1 = e_1; \ldots; x_n = e_n\ \rrbracket, \Phi, \rho_0, \sigma, \perp_{DMAP}) \\
v & = & \rho'[x_j] \\
R & = & \widehat{\mathit{Reachable}}\,(v, \sigma') \\
I & = & \displaystyle\bigcup_{y \in FV\,(e)} \widehat{\mathit{Reachable}}\,(\rho'[y], \sigma')
\end{array}
$$

Previously, I was computed with respect to σ, the incoming store, and now it is computed with respect to σ', the result store.

7.3.4 Example I-Structure Program

We will now execute an example I-structure program to see how lifetime analysis performs in the presence of side-effects. In the following I-structure program, procedure $\mathtt{f}_0$ allocates an empty

I-structure and passes it to procedure g, which fills it in with two tuples.

```
def g (a) =
  k2/ t1 = l1MakeTuple(6.823, 6.847);
     t2 = l2MakeTuple(6.847, 6.823);
     x1 = Store(a, 0, t1);
     x2 = Store(a, 1, t1);
     x3 = Store(a, 2, t2);
    in True /;

def f0 () =
  k0{ a = l0MakeIArray(3);
     v = k1g(a);
     r = Fetch(a, 0);
    in r };
```

The tuples allocated in procedure g and bound to variables t1 and t2 are not returned as part of g's result, yet they escape from the body of g. They are stored into the I-structure passed in as g's argument.

The result yielded by executing this program under the instrumented interpreter would be:

$$
\begin{array}{ll}
\langle & \epsilon.k_0.k_1.k_2 : l_1, \\
 & \left[\begin{array}{l}
\epsilon.k_0 : l_0 \rightarrow \langle_{Array}\ 3, \epsilon.k_0.k_1.k_2 : l_1, \epsilon.k_0.k_1.k_2 : l_1, \epsilon.k_0.k_1.k_2 : l_2 \rangle\ , \\
\epsilon.k_0.k_1.k_2 : l_1 \rightarrow \langle_{Tuple}\ \underline{6.847}, \underline{6.823} \rangle\ , \\
\epsilon.k_0.k_1.k_2 : l_2 \rightarrow \langle_{Tuple}\ \underline{6.847}, \underline{6.823} \rangle\ ,
\end{array} \right], \\
 & \{ \langle\ \epsilon.k_0 : l_0,\ \epsilon.k_0\ \rangle\ , \langle\ \epsilon.k_0.k_1.k_2 : l_1,\ \epsilon.k_0.k_1.k_2\ \rangle\ , \langle\ \epsilon.k_0.k_1.k_2 : l_2,\ \epsilon.k_0.k_1.k_2\ \rangle \}, \\
 & \emptyset, \\
 & \{ \langle\ \epsilon.k_0 : l_0,\ \epsilon.k_0\ \rangle\ , \langle\ \epsilon.k_0 : l_0,\ \epsilon.k_0.k_1.k_2\ \rangle \}\ \rangle
\end{array}
$$

The abstract interpreter, using our improved activation labels, would yield the following:

$$
\begin{array}{ll}
\langle & \{ \epsilon.k_0.k_1.k_2 : l_1, \epsilon.k_0.k_1.k_2 : l_2 \}, \\
 & \left[\begin{array}{l}
\epsilon.k_0 : l_0 \rightarrow \langle_{Array}\ Shared, \{ \epsilon.k_0.k_1.k_2 : l_1, \epsilon.k_0.k_1.k_2 : l_2 \} \rangle\ , \\
\epsilon.k_0.k_1.k_2 : l_1 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N} \rangle\ , \\
\epsilon.k_0.k_1.k_2 : l_2 \rightarrow \langle_{Tuple}\ \underline{N}, \underline{N} \rangle\ ,
\end{array} \right], \\
 & \emptyset, \\
 & \perp_{DMAP}\ \rangle
\end{array}
$$

We lose some information in the abstract domain, because it appears that both tuples escape from the result of procedure f_0. This approximation is safe, because nothing that is reachable under the instrumented semantics appears unreachable under the abstracted semantics. Lifetime analysis using the abstract interpreter correctly determines that the two tuples may escape from the body of procedure g, even though neither of them is directly returned as part of g's result.

7.4 Summary

This chapter described the abstraction of the array domains and how we have to use spatial summarization to obtain a finite representation of arrays at compile-time. We used this abstract array domain to extend our lifetime analysis algorithm to handle programs containing arrays. We then extended the abstracted array domains and abstract interpreter in order to perform sharing analysis on array elements, because the compiler can generate more efficient deallocation code if it knows that no element of an array is shared.

We also added I-structure arrays to KID$^-$ in this chapter. I-structures increase the expressiveness of the language and allow us to write some programs more efficiently than if we had to use the functional `MakeArray` construct. I-structures also introduce a new path for objects to escape from a control region — objects may escape by being stored into I-structures that were inherited from the surrounding context. We showed that our lifetime analysis algorithm correctly handles this case.

Chapter 8

Algebraic and Recursive Types

In Chapters 4 and 5 we saw how to summarize the behavior of KID$^-$ over tuples, numbers, and booleans. In Chapter 7 we introduced the notion of spatial summarization which was necessary to generalize the values of arrays. Spatial summarization was introduced because, in general, the size of an array can only be determined at run-time, and we needed to be able to summarize the behavior of a program over all possible arrays.

In this chapter, we develop an abstraction of algebraic types. The abstraction of non-recursive algebraic types is very straightforward. This abstraction is discussed in the first section of this chapter.

The abstraction of recursive algebraic types in many abstract interpreters is very difficult because the size of the representation of a recursively typed object can grow without bound. We see in the second section of this chapter; however, that our abstract interpreter does not suffer from this problem.

Our abstraction of non-recursive algebraic types is adequate for recursive algebraic types as well. This abstraction involves a form of spatial summarization because the number of nodes composing an object of recursive type can only be known at run-time, but our abstraction compresses it into a finite number of nodes at compile-time.

Although our abstraction of algebraic types is general enough to model any recursive algebraic type safely, the only recursive type for which our implementation of the deallocation code insertion algorithm can generate deallocation code is lists. We discuss our abstraction of lists in the third section of this chapter and compare our abstraction with that of other researchers.

The spatial summarization that occurs in the abstraction of recursive types makes it difficult to insert code to deallocate these objects because there may be sharing between the nodes of the objects. We need a better idea about the sharing that occurs between the elements of a recursively typed object. We discuss a way to approach this problem in Section 11.1.3.

8.1 Abstraction of Algebraic Types

In Chapter 2 we saw that oneofs, or algebraic types, are represented by tagged structures in the standard interpreter. The tags distinguish instances of the different disjuncts of the algebraic type.

Evaluation of a given expression in different contexts may return different disjuncts of an algebraic type. For this reason, an abstract oneof value must be a product of each of the possible disjuncts, rather than a sum, as in the instrumented interpreter. The abstracted value must capture information about the values resulting from evaluation in all possible contexts. For instance, an expression that results in an object of type `transaction`, where:

```
type transaction = deposit I | withdrawal I
```

might return either a reference to a deposit of 19.92, represented by:

$$\langle_{\mathit{0,2}}\ \underline{19.92}\rangle$$

or a withdrawal of 353.0, represented by:

$$\langle_{\mathit{1,2}}\ \underline{353.0}\rangle\,.$$

The abstract interpreter must represent both possibilities in a single value. This expression would return a reference to the following abstract oneof:

$$\langle_{\mathit{Oneof}}\ \langle_{\mathit{0}}\ \underline{N}\rangle\,,\langle_{\mathit{1}}\ \underline{N}\rangle\rangle$$

which represents either a oneof with tag 0 (a deposit) or a oneof with tag 1 (a withdrawal).

The above abstract `transaction` value is the most defined abstract transaction value. This abstract value represents standard values that are either deposits or withdrawals. We can also represent `transactions` that could only be deposits as follows:

$$\langle_{\mathit{Oneof}}\ \langle_{\mathit{0}}\ N\rangle\,,\ \perp\rangle$$

We represent `transactions` that can only be withdrawals as follows:

$$\langle_{\mathit{Oneof}}\ \perp,\ \langle_{\mathit{1}}\ N\rangle\rangle$$

Either or both of the components of an abstract `transaction` structure can be bottom. If an expression e evaluated in some context $\hat{C}$ under the abstract interpreter yields a `transaction` structure with bottom for the `deposit` component, then the same expression could never yield a `deposit` structure if it was evaluated under the standard interpreter in a context compatible to $\hat{C}$.

The following are the abstraction functions that map standard transactions into abstract transactions:

$$\begin{array}{rcl} ABS_{Transaction}\,(\langle_{\mathit{0,2}}\ n\rangle) & = & \langle_{\mathit{Oneof}}\ \langle_{\mathit{0}}\ \underline{N}\rangle\,,\perp\rangle \\ ABS_{Transaction}\,(\langle_{\mathit{1,2}}\ n\rangle) & = & \langle_{\mathit{Oneof}}\ \perp,\langle_{\mathit{1}}\ \underline{N}\rangle\rangle \end{array}$$

This method of summarizing information about algebraic types is very general. As we shall see in Section 8.2, it even handles recursive types appropriately.

$$\begin{array}{l} Abs_{Oneof}\,(\langle_{tag,n}\ v_0, \cdots, v_m\rangle) = \\ \qquad \langle_{Oneof}\ \bot, \cdots, \langle_{tag}\ Abs_V\,(v_0), \cdots ABS_V\,(v_m)\rangle\,, \cdots, \bot\rangle \end{array}$$

Figure 8.1: Oneof abstraction operator

$$\begin{array}{l} \langle_{Oneof}\ d_0, \cdots, d_n\rangle \sqcup_{Oneof} \langle_{Oneof}\ d'_0, \cdots, d'_n\rangle = \\ \qquad \langle_{Oneof}\ (d_0 \sqcup_{Disjunct} d_0), (d'_n \sqcup_{Disjunct} d'_n)\rangle \\ \langle_i\ v_0, \cdots, v_m\rangle \sqcup_{Disjunct} \langle_i\ u_0, \cdots, u_m\rangle = \\ \qquad \langle_i\ (v_0 \sqcup_V u_0), \cdots, (v_m \sqcup_V u_m)\rangle \end{array}$$

Figure 8.2: Oneof least upper bound operators

$$\begin{array}{rcl} \langle_{Oneof}\ d_0, \cdots, d_n\rangle \sqsubseteq_{Oneof} \langle_{Oneof}\ d'_0, \cdots, d'_n\rangle & = & \bigwedge_i d_i \sqsubseteq_{Disjunct} d'_i \\ \langle_i\ v_0, \cdots, v_m\rangle \sqsubseteq_{Disjunct} \langle_i\ u_0, \cdots, u_m\rangle & = & \bigwedge_i v_0 \sqsubseteq_V u_0 \end{array}$$

Figure 8.3: Oneof ordering operators

8.1.1 Domains for Abstract Algebraic Types

We add the following definitions of the abstract *Disjunct* and *Oneof* domains and revise the storable value domain SV as shown:

$$\begin{array}{rcl} Disjunct & = & \langle_N\ V, \cdots, V\rangle_\bot \\ Oneof & = & \langle_{Oneof}\ Disjunct, \cdots, Disjunct\rangle \\ SV & = & (Tuple + Array + Oneof)_\bot \end{array}$$

Each value in the *Disjunct* domain is either a tagged tuple of denotable values or bottom. Each value in the *Oneof* domain is a tuple of *Disjunct*'s, and storable values (SV) are either tuples, arrays, oneofs, or bottom. Stores still map abstract object labels to storable values.

Figure 8.1 contains the function Abs_{Oneof}, which maps standard oneof values into abstract oneof values. Figure 8.2 contains the least upper bound operator on abstract oneofs, and Figure 8.3 contains the ordering operator for abstract oneofs.

8.1.2 Abstract Interpretation of Algebraic Types

Figure 8.4 contains the clauses of the abstract interpreter that evaluate the primitives that create and manipulate oneof objects. These clauses are similar to the ones from the instrumented interpreter, except that they manipulate abstract oneof values.

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^{l}\texttt{MakeOneof}_{tag,n_{tags}}(se_1, \cdots, se_m) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\quad \{ \begin{array}[t]{ll}
v_1 & = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \ ; \\
 & \vdots \ ; \\
v_{n_i} & = \mathcal{SE}_A [\![\ se_{n_i} \ ]\!] \rho \ ; \\
ol & = \alpha : l; \\
d_0 & = \perp; \\
 & \vdots \\
d_{tag} & = \langle_{tag} \ v_1, \cdots, v_m \rangle \ ; \\
 & \vdots \\
d_{n_{tags}} & = \perp; \\
v_{oneof} & = \langle_{Oneof} \ d_0, \cdots, d_{n_{tags}-1} \rangle \ ; \\
\sigma' & = \sigma[ol \rightarrow (v_{oneof} \sqcup \sigma[ol])]; \\
\multicolumn{2}{l}{\text{in } \langle \ \{ol\}, \sigma', \emptyset, \perp_{DMap} \ \rangle \ \}}
\end{array} \\
\mathcal{E}_A [\![\ \texttt{Is}_{tag}?\,(se) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\quad \langle \ \underline{b}, \sigma, \emptyset, \perp_{DMap} \ \rangle \\
\mathcal{E}_A [\![\ \texttt{Select}_{tag,i}\,(se) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\quad \{ \ ls = \mathcal{SE}_A [\![\ se \ ]\!] \rho \ ; \\
\qquad \langle_{Oneof} \ d_0, \cdots, d_{tag-1}, d_{tag}, d_{tag+1}, \cdots, d_{n_{tags}-1} \rangle = \\
\qquad\qquad\qquad\qquad \bigsqcup_{ol \in ls} \sigma[ol]; \\
\qquad \langle_{tag} \ v_1, \cdots, v_m \rangle = d_{tag}; \\
\qquad \text{in } \langle \ v_i, \sigma, \emptyset, \perp_{DMap} \ \rangle \ \}
\end{array}
$$

Figure 8.4: Abstract interpretation of algebraic type primitives

8.2 Abstraction of Recursive Types

Recursive types are a special case of algebraic types. Even though the individual nodes of a recursively typed object are of fixed size, the object can have a size that is unbounded. In the abstract interpreter, we need some form of spatial summarization that collapses a list or tree object of potentially unbounded size into a representation with bounded size. As we see later in this section, the spatial summarization of recursive types naturally follows from our abstraction of locations and stores.

Consider the definition of `copy_list`, shown below. This procedure takes a list and returns a copy of the list.

```
def copy_list(l) =
 if Nil?(l)
   then ^l0 Nil
   else { a = Hd(l);
          as = Tl(l);
          l' = ^k0 copy_list(as);
          bs = ^l1 MakeCons(a, l');
         in bs };
```

The result of a call to `copy_list` under the standard interpreter is a list whose length is the same as the length of the input list `l`.

Abstract interpreters that do not use a store or retrieval function to model structures have difficulty abstracting recursive types. Under these interpreters, the result from a call to `copy_list` would be a potentially infinite representation of a list because all procedure calls and both branches of all conditionals are evaluated. Consequently, the abstract interpretation would not terminate unless some action was taken to bound the size of the representation of the list.

There are three ways we can bound the sizes of the representations of recursive types in abstract interpretation. First, we can compress the domain *a priori*, as we did with the integer and boolean domains. Second, we can apply a generalization, or summarization, operator to such representations. Third, we can structure our domains and interpreter so that we can guarantee that no values of unbounded size are ever constructed.

8.2.1 Abstraction of Recursive Types by Domain Compression

Much of the functional language community has taken the first approach to abstracting recursive types. For instance, Wadler compresses the abstract list domain into the following four elements for strictness analysis [42]:

$$\begin{array}{ll}
\top\epsilon & \text{— any finite list, no member of which is } \perp \\
| & \\
\perp\epsilon & \text{— any finite list, some member of which is } \perp \\
| & \\
\infty & \text{— any infinite list or approximation to one, except } \perp \\
| & \\
\perp & \text{— } \perp
\end{array}$$

This list domain ensures that the abstract interpretation terminates in a finite amount of time, because all list objects have fixed size.

The list domain defined by Wadler can only capture information about uniform lists. It cannot capture information about lists that may begin with a finite sequence of `cons` cells with non-uniform properties followed by a uniform list. Furthermore, it is difficult to see how to define appropriate abstract domains for other algebraic types based on this abstraction of lists.

8.2.2 Abstraction of Recursive Types by *Ad Hoc* Object Compression

The second approach to limiting the size of the abstract representation of a recursively typed object is to apply a compression operator to the representation: the operator generalizes the abstract value

Figure 8.5: Abstract list value before and after compression

representation and limits the number of nodes in the representation to some arbitrary bound k. Figure 8.5 shows a representation of a list (a) before, and (b) after compression. The compressed abstract list contains only two nodes.

The drawback to this approach is that it is difficult to choose the bound k on the object representation's size that yields the best information for a particular program. In some cases the value of k may be too large, resulting in extra overhead during analysis but not providing more information. In other cases, the value of k is too small and useful information is obscured.

8.2.3 Abstraction of Recursive Types by Object Label Compression

The third approach, and the approach taken in this thesis, is to structure the domains and interpreter in order to guarantee the finiteness of the representation of any list or algebraic type. The use of stores (or other retrieval functions) that map a finite number of abstract object labels to abstract storable values guarantees that the size of a recursively typed object representation remains finite. All nodes with identical labels are coallesced into a single node. Thus, there will only be a finite number of distinct nodes in the representation of any object or group of objects.

We defined our abstract activation labels, object labels, denotable values, storable values and stores so that we could analyze programs containing only tuples. We then augmented the storable value domains so that we could analyze arrays and non-recursive algebraic types. With this abstraction we can also analyze programs that use recursive types because the number of distinct abstract objects in a program is bounded by the size of the object label domain. The abstract object label domain is bounded in size by the number of paths through the call-graph of a program (disregarding cycles).

8.2.4 Spatial Summarization in Recursively Typed Objects

The abstraction of recursively typed objects involves a form of spatial summarization, as did the abstraction of arrays. In arrays, we summarized a single object whose size was known only at run time by an abstract object by a single component. In recursive types, we summarize a concrete graph or tree containing an unknown number of nodes by a graph with a fixed number of abstract nodes. Any two nodes in the concrete graph whose object labels map to the same abstract object label will be summarized by a single abstract object.

For example, consider the type `tree`:

```
type tree = node tree tree | leaf N;
```

and the following value and store that represent a concrete tree object:

$$
\begin{array}{l}
\alpha : l_0, \\
\left[\begin{array}{l}
\alpha : l_0 \rightarrow \langle_{0,2}\ \epsilon.k_1 : l_0, \epsilon.k_2 : l_0\rangle\,, \\
\alpha.k_1 : l_0 \rightarrow \langle_{0,2}\ \epsilon.k_1.k_1 : l_1, \epsilon.k_1.k_2 : l_1\rangle\,, \\
\alpha.k_2 : l_0 \rightarrow \langle_{0,2}\ \epsilon.k_2.k_1 : l_1, \epsilon.k_2.k_2 : l_0\rangle\,, \\
\alpha.k_2.k_2 : l_0 \rightarrow \langle_{0,2}\ \epsilon.k_2.k_2.k_1 : l_1, \epsilon.k_2.k_2.k_2 : l_0\rangle\,, \\
\alpha.k_1.k_1 : l_1 \rightarrow \langle_{1,2}\ \underline{1}\rangle\,, \\
\alpha.k_1.k_2 : l_1 \rightarrow \langle_{1,2}\ \underline{2}\rangle\,, \\
\alpha.k_2.k_1 : l_1 \rightarrow \langle_{1,2}\ \underline{3}\rangle\,, \\
\alpha.k_2.k_2.k_1 : l_1 \rightarrow \langle_{1,2}\ \underline{4}\rangle\,, \\
\alpha.k_2.k_2.k_2 : l_1 \rightarrow \langle_{1,2}\ \underline{5}\rangle
\end{array} \right].
\end{array}
$$

This tree consists of 4 interior nodes and 5 leaf nodes.

If we abstract the activation labels appearing in this representation to the following set:

$$\{\alpha.(k_0 + k_1)^*\},$$

then the abstract tree representation collapses to the following 2-node abstract value and store:

$$
\begin{array}{l}
\{\alpha.(k_0 + k_1)^* : l_0\}, \\
\left[\begin{array}{l}
\alpha.(k_0 + k_1)^* : l_0 \rightarrow \\
\quad \left\langle_{Oneof} \left\langle_{0} \left\{ \begin{array}{l} \alpha.(k_0 + k_1)^* : l_0, \\ \alpha.(k_0 + k_1)^* : l_1 \end{array} \right\}, \left\{ \begin{array}{l} \alpha.(k_0 + k_1)^* : l_0, \\ \alpha.(k_0 + k_1)^* : l_1 \end{array} \right\} \right\rangle, \perp \right\rangle, \\
\alpha.(k_0 + k_1)^* : l_1 \rightarrow \langle_{Oneof}\ \perp, \langle_{1}\ \underline{N}\rangle\rangle
\end{array} \right].
\end{array}
$$

This representation consists of only two abstract nodes. The abstraction of activation labels is normally derived from the call-graph of a program.

We do not add anything to the abstract domains or the abstract interpreter in this section. The complexity we added in the basic framework has paid off by being general enough to handle recursively typed objects. In the following section, we describe an extension to the abstract interpreter to model lists as a special case of Oneofs.

8.3 Abstraction of Lists in KID⁻

The list type, which is a particular recursive algebraic type, could be modeled using our *Oneof* domain. However, our implementation of the deallocation command insertion algorithm generates specialized code to deallocate lists, so we model lists separately in our abstract interpreter.

$$\begin{array}{rcl} Abs_{List}\,(\langle_{Cons}\ v_0, v_1\rangle) & = & \langle_{List}\ \langle_{Cons}\ Abs_V\,(v_0), ABS_V\,(v_1)\rangle\,, \bot\rangle \\ Abs_{List}\,(\langle_{Nil}\ \rangle) & = & \langle_{List}\ \bot, \langle_{Nil}\ \rangle\rangle \end{array}$$

Figure 8.6: List abstraction operator

$$\begin{array}{rcl} \langle_{List}\ c_0, n_0\rangle \sqcup_{List} \langle_{List}\ c_1, n_1\rangle & = & \langle_{List}\ (c_0 \sqcup_{Cons} c_1), (n_0 \sqcup_{Nil} n_1)\rangle \\ \langle_{Cons}\ v_0, v_1\rangle \sqcup_{Cons} \langle_{Cons}\ w_0, w_1\rangle & = & \langle_{Cons}\ (v_0 \sqcup_V w_0), (v_1 \sqcup_V w_1)\rangle \\ \langle_{Nil}\ \rangle \sqcup_{Nil} \langle_{Nil}\ \rangle & = & \langle_{Nil}\ \rangle \end{array}$$

Figure 8.7: List least upper bound operators

$$\begin{array}{rcl} \langle_{List}\ c_0, n_0\rangle \sqsubseteq_{List} \langle_{List}\ c_1, n_1\rangle & = & (c_0 \sqsubseteq_{Cons} c_1) \wedge (n_0 \sqsubseteq_{Nil} n_1) \\ \langle_{Cons}\ v_0, v_1\rangle \sqsubseteq_{Cons} \langle_{Cons}\ w_0, w_1\rangle & = & (v_0 \sqsubseteq_V w_0) \wedge (v_1 \sqsubseteq_V w_1) \\ \langle_{Nil}\ \rangle \sqsubseteq_{Nil} \langle_{Nil}\ \rangle & = & \texttt{True} \end{array}$$

Figure 8.8: List ordering operators

8.3.1 Abstract List Domains

The definition of the abstracted list domain that we use follows:

$$\begin{array}{lcll} v_{list} \in List & = & \langle_{List}\ \langle_{Cons}\ V, Ls\rangle_\bot\,, \langle_{Nil}\ \rangle_\bot\rangle & \text{Lists} \\ sv \in SV & = & (Tuple + Array + Oneof + List)_\bot & \text{Storable Values} \end{array}$$

Abstract lists, like abstract oneofs, are represented by a pair of tagged disjuncts. If one of these components is bottom, that indicates that none of the concrete values represented by this abstract list could evaluate to `Cons` or `Nil`. If both of these are non-bottom, then the corresponding standard values could be either `Cons` or `Nil`.

Figure 8.6 contains the function Abs_{List}, which maps standard list values into abstract list values. Figure 8.7 contains the least upper bound operator on abstract lists, and Figure 8.8 contains the ordering operator for abstract lists.

This list abstraction is safe, in that it preserves the reachability of the list elements. Abstract list representations may suffer from spatial summarization and lose information about whether the `cons` cells are shared. If a list is constructed from distinct invocations of `Cons`, then the analysis will obtain complete information (as with tuples). However, if a list is constructed by a recursive procedure, then the calls to `Cons` will not have distinct activation labels, and a cyclic representation will be constructed.

The compiler must assume that any list whose abstract representation is cyclic may represent a cyclic list. The compiler must also assume that the objects pointed to by a cyclic abstract list may be shared. Therefore, if the compiler inserts code to deallocate a list, it must ensure that each distinct `cons` cell in the list is deallocated only once.

$$
\begin{array}{rcl}
\mathcal{E}_A[\![\ {}^l\texttt{Cons}\,(se_1, se_2)\ ]\!]\rho\sigma\alpha\Phi & = & \{\ v_1 \ \ = \mathcal{SE}_A[\![\ se_1\ ]\!]\rho\,;\\
 & & \ \ v_2 \ \ = \mathcal{SE}_A[\![\ se_2\ ]\!]\rho\,;\\
 & & \ \ v_{list} = \langle_{List}\ \langle_{Cons}\ v_1, v_2\rangle\,, \perp\rangle\,;\\
 & & \ \ ol \ \ = \alpha : l;\\
 & & \ \ v'_{list} = \sigma[ol];\\
 & & \ \ \sigma' \ \ = \sigma[ol \rightarrow (v_{list} \sqcup v'_{list})];\\
 & & \ \ \text{in}\ \langle\ \{ol\}, \sigma', \emptyset, \perp_{DMAP}\ \rangle\ \}\\
\mathcal{E}_A[\![\ \texttt{Hd}\,(se)\ ]\!]\rho\sigma\alpha\Phi & = & \{\ ls = \mathcal{SE}_A[\![\ se\ ]\!]\rho\,;\\
 & & \ \ \langle_{List}\ \langle_{Cons}\ v_1, v_2\rangle\,, \langle_{Nil}\ \rangle\rangle =\\
 & & \qquad \langle_{List}\ \langle_{Cons}\ \perp, \perp\rangle\,, \perp\rangle \sqcup \bigsqcup_{ol\in ls} \sigma[ol];\\
 & & \ \ \text{in}\ \langle\ v_1, \sigma, \emptyset, \perp_{DMAP}\ \rangle\ \}\\
\mathcal{E}_A[\![\ \texttt{Tl}\,(se)\ ]\!]\rho\sigma\alpha\Phi & = & \{\ ls = \mathcal{SE}_A[\![\ se\ ]\!]\rho\,;\\
 & & \ \ \langle_{List}\ \langle_{Cons}\ v_1, v_2\rangle\,, \langle_{Nil}\ \rangle\rangle =\\
 & & \qquad \langle_{List}\ \langle_{Cons}\ \perp, \perp\rangle\,, \perp\rangle \sqcup \bigsqcup_{ol\in ls} \sigma[ol];\\
 & & \ \ \text{in}\ \langle\ v_2, \sigma, \emptyset, \perp_{DMAP}\ \rangle\ \}\\
\mathcal{E}_A[\![\ {}^l\texttt{Nil}\,()\ ]\!]\rho\sigma\alpha\Phi & = & \{\ v_{list} = \langle_{List}\ \perp, \langle_{Nil}\ \rangle\rangle\,;\\
 & & \ \ ol \ \ = \alpha : l;\\
 & & \ \ v'_{list} = \sigma[ol];\\
 & & \ \ \sigma' \ \ = \sigma[ol \rightarrow (v_{list} \sqcup v'_{list})];\\
 & & \ \ \text{in}\ \langle\ \{ol\}, \sigma', \emptyset, \perp_{DMAP}\ \rangle\ \}\\
\mathcal{E}_A[\![\ \texttt{Nil?}\,(se)\ ]\!]\rho\sigma\alpha\Phi & = & \langle\ \underline{B}, \sigma, \emptyset, \perp_{DMAP}\ \rangle
\end{array}
$$

Figure 8.9: Abstracted evaluation of list primitives

8.3.2 Additions to Abstract Interpreter

Figure 8.9 contains the evaluation rules for list primitives in the abstract interpreter. They are similar to the instrumented evaluation rules, except that labels are abstracted and lists are products of `Cons` and `Nil` objects.

8.3.3 Representative List Inputs

To use our deallocation command safety verification algorithm and deallocation command insertion algorithm, we must have a method to construct representative input values for objects of recursive types. This section defines the clause of procedure $\mathcal{CV}$ that constructs representative list arguments.

The inputs we pass as inputs to procedures under analysis must be of the correct type and must be detectable wherever they are passed within the procedure. This constrains the abstract list values that we may use as representative inputs to a function expecting a list. We cannot tell *a priori* how many of the `cons` cells in a list input are dereferenced by a procedure; consequently, the abstract list must either have infinitely many `cons` cells or be circular. We use circular list representations as representative inputs. The clause of procedure $\mathcal{CV}$ that constructs representative list values, shown below, returns a circular abstract list whose head contains an element of the correct type.

$$\begin{array}{lcl} \mathcal{CV}[\![\ List\ \tau_1\ ]\!] & = & \{\ \langle\ v,\sigma'\ \rangle = \mathcal{CV}[\![\ \tau_1\ ]\!]\,; \\ & & \quad l \qquad\quad = Newloc\,(); \\ & & \quad \sigma'' \qquad = \sigma'[l \rightarrow \langle_{List}\ \langle_{Cons}\ v, \{l\}\rangle, \langle_{Nil}\ \rangle\rangle]; \\ & & \quad \text{in}\ \langle\ \{l\}, \sigma''\ \rangle\ \} \end{array}$$

The rationale for the safety of using circular lists as inputs is that any particular list under the standard interpreter composed of a finite number of `cons` cells labeled l_0 through l_n and containing values v_0 through v_n in the heads of each `cons` can be *contained* by a cyclic abstract list. We say list representation r_0 *contains* r_1 if ($r_0 \sqsubseteq_A bsr_1$). The abstraction of such a list is contained by the following abstract value-store pair:

$$\langle\ \{l_0,\cdots,l_n\},\ \perp_{Store}\left[\begin{array}{c} l_0 \rightarrow \langle_{List}\ \langle_{Cons}\ \widehat{v}, \{l_0,\cdots,l_n\}\rangle, \langle_{Nil}\ \rangle\rangle \\ \vdots \\ l_n \rightarrow \langle_{List}\ \langle_{Cons}\ \widehat{v}, \{l_0,\cdots,l_n\}\rangle, \langle_{Nil}\ \rangle\rangle \end{array}\right]\rangle,$$

where $\widehat{v}$ is the least upper bound of the elements of the concrete list: v_0 through v_n and $\{l_0,\cdots,l_n\}$ is the abstraction of the locations where the concrete list resides.

Therefore, we can show that the analysis of a function using a circular abstract list representation is safe for any list to which the function is actually applied. We show this by substituting the set of labels of the `Cons` cells in the actual list for the label of the `Cons` in the circular list and substituting the least upper bound of the elements of the actual list for the element of the representative list.

8.3.4 List Examples

Now let us examine a couple of examples that use lists, in order to see what information we can gather and how far we can go with them. Procedure `scale_list` takes a number and a list of numbers and returns a new list of scaled numbers; procedure `inc_list` takes a number and a list of numbers and returns a list of incremented numbers.

```
def scale_list (s, l) =
  if nil?(l) then ^l0 nil
  else { x = hd(l);
         xs = tl(l);
         x' = s * x;
         sl = ^k0 scale_list(s, xs);
         r = ^l1 Cons(x', sl);
        in r };

def inc_list (delta, l) =
  if nil?(l) then ^l2 nil
  else { x = hd(l);
         xs = tl(l);
         x' = delta +  x;
         sl = ^k1 inc_list(s, xs);
         r = ^l3 Cons(x', sl);
        in r };
```

The representative input vector for scale_list is

$$\langle\ \underline{N}, \{l_{-1}\},\ \bot_{Store}[l_{-1} \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_{-1}\}\rangle, \langle_{Nil}\ \rangle\rangle]\ \rangle$$

This represents a number and a list whose head is a number and whose tail is either `nil` or the list itself.

The value returned from `scale_list` is another list, consisting of a `cons` in location l_1 or `nil` in location l_0:

$$\langle\ \{l_0, l_1\},\ \bot_{Store}\left[\begin{array}{l} l_{-1} \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_{-1}\}\rangle, \langle_{Nil}\ \rangle\rangle \\ l_0 \rightarrow \langle_{List}\ \bot, \langle_{Nil}\ \rangle\rangle \\ l_1 \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_0, l_1\}\rangle, \bot\rangle \end{array}\right]\rangle$$

From this value we can determine that the list passed into function `scale_list` cannot be reached from its result. Furthermore, we know that the result is a list that must have been allocated within the call to `scale_list`.

We obtain similar results when we analyze procedure `inc_list`. What is the behavior if we compose these two functions, as in procedure `inc_scale_list`?

```
def inc_scale_list (delta, s, x) =
  { x' = k2 scale_list(s, x);
    r = k3 inc_list(delta, x');
   in r };
```

If we evaluate the bindings in the body of `inc_scale_list` when applied to the following input vector:

$$\langle\ \underline{N}, \underline{N}, \{l_{-2}\},\ \bot_{Store}[l_{-2} \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_{-2}\}\rangle, \langle_{Nil}\ \rangle\rangle]\ \rangle$$

we obtain the following values:

$$\begin{array}{rcl} \rho' & = & \left[\begin{array}{l} delta \rightarrow \underline{N} \\ s \rightarrow \underline{N} \\ x \rightarrow \{l_{-2}\} \\ x' \rightarrow \{l_0, l_1\} \\ r \rightarrow \{l_2, l_3\} \end{array}\right] \\ \sigma' & = & \left[\begin{array}{l} l_{-2} \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_{-2}\}\rangle, \langle_{Nil}\ \rangle\rangle\ l_0 \rightarrow \langle_{List}\ \bot, \langle_{Nil}\ \rangle\rangle \\ l_1 \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_0, l_1\}\rangle, \bot\rangle \\ l_2 \rightarrow \langle_{List}\ \bot, \langle_{Nil}\ \rangle\rangle \\ l_3 \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_0, l_1\}\rangle, \bot\rangle \end{array}\right] \\ I & = & \{l_{-2}\} \\ R & = & \{l_2, l_3\} \end{array}$$

The set I of labels reachable from the body of `inc_scale_list` contains l_{-2}, and R, the set of labels reachable from the result of `inc_scale_list`'s body contains l_2 and l_3. From this we can conclude that the object to which x' is bound, consisting of locations l_0 and l_1, must have been allocated within the body of `inc_scale_list` and that this object does not escape from there. Consequently, we can insert a deallocation command for variable x'.

What should this deallocation command do? The whole list that has been allocated is garbage, but we cannot determine the size of the list from the abstract values. Nor can we determine if there is any sharing in the list. We can insert code to deallocate this whole list as long as the code checks that it never deallocates the same cons cell twice along the spine of the list.

Consider the following example, which constructs a circular list:

```
def cyclic_list (elt) =
  { r = l4Cons(elt, r);
   in r };
```

Procedure cyclic_list, applied to $\underline{N}$, returns the value

$$\langle\ \{l_4\},\ \perp_{Store}[l_4 \rightarrow \langle_{List}\ \langle_{Cons}\ \underline{N}, \{l_4\}\rangle\ , \perp\rangle]\ \rangle$$

Because the nil component of the list in location l_4 is bottom, we can conclude that this list never has a null tail — it is either infinite or cyclic. However, we still cannot tell how many cons cells the list will have under the standard interpreter.

Whenever we determine that the lifetime of a list is bounded by some control region, we will insert code that recursively deallocates all distinct cons cells of the list upon termination of that control region. In Chapter 10, we discuss the run-time performance of the code that deallocates potentially cyclic lists compared to the code that deallocates acyclic lists.

Chapter 9

Higher-Order Functions

Many modern programming languages have higher-order functions. That is, one can pass procedures around as values. Procedures can take procedures as arguments and return procedures as values. This ability to pass procedures around provides a great deal of flexibility in writing programs.

Unfortunately, many approaches to lifetime analysis that use abstract interpretation do not model higher-order functions. One of the main difficulties in the abstraction of procedure values is how to take the least upper bound of two functions. The least upper bound is well-defined theoretically as long as the two functions have the same domains and ranges. If we have functions f_0 and f_1, then the least upper bound can be defined as a new function:

$$f_0 \sqcup f_1 \equiv \lambda x.(f_0(x) \sqcup f_1(x))$$

However, this definition is not always conducive to an implementation. The key here is to separate out the text of the function from the object being passed around as a value. The approach taken in this thesis is to represent functions as closures which consist of the name of a function and the values the function is closed over. The name of a function points to the text of the function — its definition in the program. We allow a prefix of a function's arguments to be provided in a closure of the function (a partial application), and the rest must be provided when the closure is applied.

The second major difficulty in the abstraction of procedure values is that the domain of functions of a given type is infinite, and so it is no longer possible to enumerate the input-output behaviors of a function that takes procedures as arguments over all possible procedures in the domain. We limit the domains of functions to contain only closures of functions that are defined in the program. There can only be finitely many functions defined in a program, and only finitely many points where those functions are closed over values.

In this chapter, we see how to add higher-order functions to KID$^-$ and to improve the abstraction of activation labels. In the first two sections we discuss the implementation of higher-order functions in the KID$^-$interpreters. In the final section we present an analysis example using higher-order functions.

9.1 Higher-Order Functions in the Instrumented Interpreter

In this section, we discuss the changes that need to be made to the domains and interpreters in order to add higher-order functions to KID$^-$. We do this for the instrumented and abstracted interpreters. In order to add higher-order functions, we need to add primitives to the language to create and apply function values, and we need to add value domains for representing function values.

9.1.1 The Closure Domain

A higher order function consists of the text of a procedure plus the lexical environment in which the function was defined. Higher order functions are most interesting when these functions can be defined in lexical environments other than the global environment. Rather than extending KID$^-$ with nested function definitions; however, we preserve the flat structure of definitions, and introduce a primitive that binds together a particular procedure definition and values from the desired lexical environment. We represent functions as *closures*. Closures are a new kind of storable value.

$$\begin{array}{rcll} cls \in Cls & = & \langle_{Cls}\ F, V, \cdots, V\rangle & \text{Closure} \\ sv \in SV & = & Tuple + Array + List + Cls & \text{Storable Values} \end{array}$$

A closure consists of a tuple of a procedure name f_i and n values. If a function f_i has r arguments, then a closure of f_i over n values, where $n < r$, may be applied to exactly $(r - n)$ values.

9.1.2 Instrumented Interpretation of Closure Primitives

We add two primitives to KID$^-$ for creating and manipulating closures: $\texttt{MakeClosure}_{f_i}$ which closes the procedure named f_i over some set of argument values, and `Apply`, which applies a closure to a set of values. We are not supporting currying directly with these primitives. The compiler can generate a sequence of intermediate functions that use `MakeClosure` and `Apply` to implement currying. This is described fully in Hochheiser [21].

The primitive `MakeClosure` is subscripted with the name f_i of the function being closed, and takes n values over which f_i is being closed. `MakeClosure` is similar to the `MakeTuple` primitive in that the expression label is used to construct a unique label ol of the structure being allocated. Note also that the set of allocation events is augmented to show that ol was allocated in the current activation α.

$$\begin{array}{l} \mathcal{E}_I [\![\ {}^{l}\texttt{MakeClosure}_{f_i}(se_1, \cdots, se_n) \ ]\!] \rho\sigma\alpha \ = \\ \qquad \{ \ v_1 = \mathcal{SE} [\![\ se_1 \ ]\!] \rho \ ; \\ \qquad\qquad \vdots \\ \qquad \ \ v_n = \mathcal{SE} [\![\ se_n \ ]\!] \rho \ ; \\ \qquad \ \ ol \ = \alpha : l; \\ \qquad \ \ \sigma' = \sigma[ol \rightarrow \langle_{Cls}\ f_i, v_1, \cdots, v_n\rangle]; \\ \qquad \ \ \text{in } \langle \ ol, \sigma', \{\langle \ ol, \ \alpha \ \rangle\}, \emptyset, \emptyset \ \rangle \ \} \end{array}$$

The primitive `Apply` takes as its first argument a closure of a function f_i with arity r and n values over which the function is closed. There must be $(r-n)$ more values supplied to `Apply`, so that it can make a full-arity application of function f_i. This primitive is similar to user function applications. First, we evaluate the arguments and dereference the closure from the incoming store. Then we evaluate the body of the closed function f_i in that activation α' and the proper environment, constructed from the values from the closure and from the inputs to `Apply`.

$$
\begin{array}{l}
\mathcal{E}_I [\![\ {}^k\texttt{Apply}\,(se, se_{n+1}, \cdots, se_r) \ ]\!] \rho\sigma\alpha \ = \\
\quad \begin{array}{ll}
\{ \ ol & = \mathcal{SE} [\![\ se \ ]\!] \rho \ ; \\
\ \ v_{n+1} & = \mathcal{SE} [\![\ se_{n+1} \ ]\!] \rho \ ; \\
 & \ \vdots \\
\ \ v_r & = \mathcal{SE} [\![\ se_r \ ]\!] \rho \ ; \\
\ \ \langle_{Cls} \ f_i, v_1, \cdots, v_n \rangle & = \sigma[ol]; \\
\ \ \alpha' & = \alpha.k; \\
\ \ \langle \ v, \sigma', \Delta^+, \Delta^-, \Delta^R \rangle & = \mathcal{E}_I [\![\ e_i \ ]\!] (\rho[v_1/x_1, \cdots, v_n/x_n, v_{n+1}/x_{n+1}, \cdots, v_r/x_r])\sigma\alpha' \ ; \\
\ \ \Delta^{R'} & = \Delta^R \cup \{ \langle \ ol, \ \alpha \ \rangle \}; \\
\ \ \text{in} \ \langle \ v, \sigma', \Delta^+, \Delta^-, \Delta^{R'} \ \rangle \ \} &
\end{array} \\
\quad \text{where } f_i(x_1, \cdots, x_r) = e_i \text{ is a definition in the program}
\end{array}
$$

We return a reference event for the closure object ol in activation α.

9.2 Higher-Order Functions in the Abstract Interpreter

This section defines the abstract closure domains and the clauses of the abstract interpreter that interpret the `MakeClosure` and `Apply` primitives.

9.2.1 Abstracting The Closure Domain

Abstraction of the closure domain is rather straightforward. We do not attempt to abstract the code text of a closure. Rather, we generalize a closure to the set of possible closures that it could be. This abstraction fits in nicely with our abstraction of storable values: a reference to the abstraction of a closure is a set of abstract object labels, each of which refers to an abstract closure. An abstract closure storable value consists of a single function name and a tuple of values over which that function is closed. The number of components in the tuple must be less than the number of arguments that the function takes.

$$
\begin{array}{lcll}
v \in V & = & (N + B + Ls)_\perp & \text{Denotable values} \\
cls \in Cls & = & \langle_{Cls} \ F, V, \cdots, V \rangle & \text{Abstract Closure} \\
sv \in SV & = & Tuple + Array + List + Cls & \text{Storable Values}
\end{array}
$$

An application of a reference to a set of abstract closures has to return the least upper bound of the values returned by applying each of the abstract closures to the abstract argument values.

In addition to abstracting the closure domain, we must choose a domain of activation labels. We have seen two choices for AL so far, the simple one from Chapter 4 and the more detailed one from Chapter 6. The more detailed abstraction requires the knowledge of the complete call graph in order to define a function $\mathcal{NAL}$ that takes an abstract activation label and an expression label and returns a new abstraction label. We cannot compute the call-graph of a program that uses higher-order functions statically, because the names of the functions that will be invoked by an application primitive are not known, in general, until run-time.

$$
\begin{aligned}
&Abs_{Cls}(\langle_{Cls}\, f_i, v_1, \cdots, v_n\rangle) = \\
&\qquad \langle_{Cls}\, f_i, Abs_V(v_0), \cdots, ABS_V(v_n)\rangle
\end{aligned}
$$

Figure 9.1: Closure abstraction operator

For programs with higher-order functions, we use a simpler definition of the AL domain and a simple function $\mathcal{NAL}$ to compute the next activation label. The definition of the domain, shown below, is the same as the AL domain used in the standard and instrumented interpreters.

$$AL \;=\; \epsilon \mid AL.L$$

However, the next activation label function guarantees that the set of activation labels remains finite. The set of activation labels is finite except for recursive functions; so $\mathcal{NAL}$ treats the activation labels of recursive function calls specially:

$$\mathcal{NAL}(\alpha, k) \;=\; \begin{cases} \alpha'.k & \text{if } \alpha = \alpha'.k.\beta \\ \alpha.k & \text{otherwise} \end{cases}$$

The motivation for this definition of $\mathcal{NAL}$ is that the activation labels of procedures that are called recursively will contain repeated expression labels. Under the standard interpreter, the next activation label from $\alpha'.k.\beta$ given expression label k would be:

$$\alpha = \alpha'.k.\beta.k$$

so the function $\mathcal{NAL}$ limits this to one occurrence of k:

$$\alpha = \alpha'.k$$

by eliding the sequence of expression labels: $k.\beta$. Note that β is empty for singly recursive functions, and it is non-empty for functions that contain multiple recursive calls or for groups of mutually recursive functions.

An alternative definition of $\mathcal{NAL}$ which may be more desirable because it further restricts the size of the activation label domain is defined below.

$$\mathcal{NAL}(\alpha, k) \;=\; k$$

This definition may yield detailed enough activation labels for most purposes.

Of course, one could use the original definition of abstract activation labels from Chapter 4, which corresponds to the following definition of $\mathcal{NAL}$.

$$\mathcal{NAL}(\alpha, k) \;=\; \epsilon$$

This definition yields the smallest possible domain of activation labels.

Figure 9.1 contains the function Abs_{Cls}, which maps standard closure values into abstract closure values. Figure 9.2 contains the least upper bound operator on abstract closures, and Figure 9.3 contains the ordering operator for abstract closures.

$$\langle_{Cls}\, f, v_1, \cdots, v_n \rangle \sqcup_{Cls} \langle_{Cls}\, g, w_1, \cdots, w_n \rangle =$$
$$\begin{cases} \langle_{Cls}\, f, (v_1 \sqcup_V w_1), \cdots, (v_n \sqcup_V w_n) \rangle & \text{if } f = g \\ \top & \text{otherwise} \end{cases}$$

Figure 9.2: Closure least upper bound operators

$$\left\langle_{Cls}\, f, v_1, \cdots, v_{n_f} \right\rangle \sqsubseteq_{Cls} \langle_{Cls}\, g, w_1, \cdots, w_{n_g} \rangle =$$
$$\begin{cases} \bigwedge\limits_i (v_i \sqsubseteq_V w_i) & \text{if } f = g \text{ and } n_f = n_g \\ \top & \text{otherwise} \end{cases}$$

Figure 9.3: Closure ordering operators

9.2.2 Termination of Abstract Interpretation

The KID$^-$type system guarantees that all closures that are created are of finite depth, where the maximum depth can be fixed at compile-time. This fact, plus the fact that there are only a fixed number of procedure texts and `MakeClosure` expressions in a given program, guarantees that there can be only a finite number of possible values for any given abstract closure arising during the abstract interpretation of a program. Thus, abstract interpretation of a program still takes a finite number of iterations to compute the function environment.

9.2.3 Abstract Interpretation of Closure Primitives

We also add the clauses to the abstract interpreter for the two primitives that create and manipulate closures: $\texttt{MakeClosure}_{f_i}$ which closes the procedure named f_i over some set of argument values, and `Apply`, which applies a closure to a set of values. The clause for `MakeClosure`, shown in Figure 9.4, uses the static expression label l alone as the object label of the allocated closure.

The clause for `Apply`, shown in Figure 9.5, first interprets the first argument to yield a set ls of references to abstract closures. The result is the least upper bound of the result of invoking each of these closures with the arguments supplied to `Apply` as well as the values carried in the closures. Each abstract closure is invoked by determining the function name f_i and the argument values, and then looking up the entry for f_i and those values in the function environment Φ. In addition, these values are added to the domain map $\Delta^{\mathcal{D}'}$ for function f_i.

9.2.4 Analysis of Higher-Order Programs

Now that we have seen how to extend the abstract domains and the abstract interpreter in order to handle higher-order functions, let us see how this affects analysis of programs containing higher-order functions. There are a number of ways this can affect the analysis and transformation of such programs. It can cause loss of information, because we have less idea what computation will be performed by an expression. It can also cause added complexity in the analysis, because it is harder to construct representative input values. But, by exposing a higher-order function as a

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^l\texttt{MakeClosure}_{f_i}(se_1, \cdots, se_n) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\qquad \{ \begin{array}[t]{ll}
v_1 & = \mathcal{SE}_A [\![\ se_1 \ ]\!] \rho \, ; \\
 & \ \vdots \\
v_n & = \mathcal{SE}_A [\![\ se_n \ ]\!] \rho \, ; \\
v_{cls} & = \langle {}_{Cls} \ f_i, v_1, \cdots, v_n \rangle \, ; \\
ol & = \alpha : l; \\
\sigma' & = \sigma[ol \rightarrow \sigma[ol] \sqcup v_{cls}]; \\
\multicolumn{2}{l}{\text{in } \langle \ \{ol\}, \sigma', \emptyset, \bot_{DMAP} \rangle \ \}}
\end{array}
\end{array}
$$

Figure 9.4: Abstract evaluation of the closure constructor

$$
\begin{array}{l}
\mathcal{E}_A [\![\ {}^k\texttt{Apply}(se, se_{n+1}, \cdots, se_r) \ ]\!] \rho\sigma\alpha\Phi \ = \\
\qquad \{ \begin{array}[t]{ll}
ls & = \mathcal{SE}_A [\![\ se \ ]\!] \rho \, ; \\
v_{n+1} & = \mathcal{SE}_A [\![\ se_{n+1} \ ]\!] \rho \, ; \\
 & \ \vdots \\
v_r & = \mathcal{SE}_A [\![\ se_r \ ]\!] \rho \, ; \\
\alpha' & = \mathcal{NAL}(\alpha) \, k; \\
\langle \ v', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}\prime} \rangle & = \bigsqcup\limits_{ol \in ls} \ \begin{array}[t]{l}
\{ \ \langle {}_{Cls} \ f_i, v_1, \cdots, v_n \rangle = \sigma[ol]; \\
\ \ \langle \ v', \sigma', \Delta^{-\prime} \ \rangle = \\
\qquad\qquad \Phi[f_i][\langle v_1, \cdots, v_n, v_{n+1}, \cdots, v_r, \sigma, \alpha' \rangle]; \\
\ \ \Delta^{\mathcal{D}\prime}[f_i] = \{\langle v_1, \cdots, v_n, v_{n+1}, \cdots, v_r, \sigma, \alpha' \rangle\}; \\
\ \ \text{in } \langle \ v', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}\prime} \rangle \ \}
\end{array} \\
\multicolumn{2}{l}{\text{in } \langle \ v', \sigma', \Delta^{-\prime}, \Delta^{\mathcal{D}\prime} \rangle \ \}}
\end{array} \\
\quad \text{where each } f_i(x_1, \cdots, x_r) = e_i \text{ is a definition in the program}
\end{array}
$$

Figure 9.5: Abstract evaluation of closure application

closure — a data structure — we have enabled the compiler to perform storage management on closures themselves.

We may lose information about the lifetime of an object created within a procedure if it is passed to a higher-order function because we may have to make worst case assumptions about the behavior of the function passed as an argument.

In the algorithms described in Chapter 5, we began computation of the function environment by computing the value of the application of each function in the program to a representative input value. What representative values should we use for functions which take closures as input? What values should we use when analyzing the body of a function and verifying or inserting deallocation commands?

During the analysis of a function body, we really do want to pass in some function value that captures the behavior of any function that could be passed in at run-time. Either we use the least upper bound of all possible values that arise under abstract interpretation, or we make a worst-case assumption about the behavior of the function. This value must satisfy the constraints of the type system, but all input values could conceivably be carried in the result of the application. Even worse, if I-structures or other side-effecting operations are supported, all input values (of the right type) could be side-effected or stored off in some structure reachable from the input values.

The approach of constructing representative input values for higher-order inputs to functions only pays off it it allows us to manage the storage of closures. Otherwise, we may as well use the least upper bound of all possible values that could be passed as input to this function.

If we look at the whole program, then we can actually determine the types of all the closures created in the program (assuming monomorphic typing), and use the set of all closures of the correct type as the input value to a function that takes closures as arguments. This process may be equivalent to taking the least upper bound of all possible inputs to a function that arise in the abstract interpretation described above, and analyzing the function when applied to this least upper bound. This process is similar to the behavior of *collecting interpreters* [24, 44].

It seems that it is better to use the most general function value that could ever be passed as input to a procedure during the analysis of that procedure than to construct representative closure values. We are likely to lose too much information if we use worst-case representative closure values rather than the closure values that arise during abstract interpretation.

9.3 Example of Abstract Interpretation of Higher-Order Functions

Let us consider the abstract interpretation of the following program. In the main procedure $\mathtt{f}_0$, one of two higher-order functions is called depending on the value of predicate `p`. What is the behavior of this program under the abstract interpreter?

```
{
  def f0 () =
    { p = e0;
      f = if p
             then l0MakeClosure_foo(10)
             else  l1MakeClosure_bar(True, 3);
      z = e1;
      r = k0Apply(f, z)
     in r };

  def foo (n,m) =
    l2MakeTuple(n,n+m);

  def bar (b,n,m) =
    { x = if b then 3 else 4;
       t = l3MakeTuple(x,n-m);
     in t };
}
```

We would like to know what the result of the invocation of function $\mathtt{f}_0$ is under the abstract interpreter. The function, or closure, to which variable `f` is bound is dependent on the value of variable `p`, a run-time value. Therefore, we must abstract the behavior of `f` over all executions.

If we evaluate the bindings of the `letrec` block in the body of $\mathtt{f}_0$, we get the following environment and store:

$$
\begin{aligned}
\rho' &= \bot_{Env}\begin{bmatrix} p & \rightarrow & \underline{B} \\ f & \rightarrow & \{l_0, l_1\} \\ z & \rightarrow & \underline{N} \\ r & \rightarrow & \{l_2, l_3\} \end{bmatrix} \\
\sigma' &= \bot_{Store}\begin{bmatrix} l_0 & \rightarrow & \langle_{Cls}\ \mathtt{foo}, \underline{N}\rangle \\ l_1 & \rightarrow & \langle_{Cls}\ \mathtt{bar}, \underline{B}, \underline{N}\rangle \\ l_2 & \rightarrow & \langle_{Tuple}\ \underline{N}, \underline{N}\rangle \\ l_3 & \rightarrow & \langle_{Tuple}\ \underline{N}, \underline{N}\rangle \end{bmatrix} \\
\Delta^{-\prime} &= \emptyset
\end{aligned}
$$

We can see by examining ρ' and σ' that `f` can be bound to a value which is either a closure of `foo` over a number or a closure of `bar` over a boolean and a number. In order to obtain the value of variable `r`, the interpreter had to evaluate the application of `foo` applied to two numbers and the application of `bar` applied to a boolean and two numbers.

Chapter 10

Performance Analysis

This chapter discusses the performance of an implementation of the analysis and transformations described in this thesis. The first section discusses our implementation of the verification and insertion algorithms, and Monsoon [36], the machine on which we ran our benchmarks. The second section presents the experiments themselves. The third section presents an optimization that generates code to allocate structures in procedure activation frames whenever possible and discusses how this affects the run-time performance of programs. The fourth section describes the difficulty of deallocating structures that may pass through zero-tripping loops, loops that execute zero or more times. The fourth section also describes a code generation strategy that can solve this problem. The fifth section describes an optimization that hoists matched allocation and deallocation commands out of loops in order to reduce the run-time overhead of storage management.

10.1 Implementation Details

Most of the theory developed in this thesis has actually been put into practice. We have an implementation of the abstract interpreter, the deallocation command verification algorithm, and the deallocation command insertion algorithm. This section describes the details of our implementation and the structure of the experiments we used to determine the overall effectiveness of our methods.

10.1.1 Implementation of the Verification and Insertion Algorithms

Our implementation of the deallocation command verification and insertion algorithms handles tuples, arrays, algebraic types, lists, and I-structures as well as a number of scalar types: booleans, integers, floating point numbers, characters, and symbols. The implementation uses activation labels similar to those described in Chapter 6, but higher-order functions are not supported. The implementation handles conditionals, loops, and the limited form of barriers shown in this thesis. The current implementation of the compiler does not insert conditional deallocation commands, but it does attempt to get complete coverage of deallocatable structures using a greedy algorithm and a careful ordering of the identifiers whose values may be deallocated.

The deallocation command verification and insertion tools are implemented as two new modules in the Id Compiler [40]. Both modules operate on *program graphs*, which are basically a dataflow graph representation of KID^-. The first module computes the function environment for the whole program and verifies and annotates each function definition. The second module walks over the program again, and actually inserts deallocation commands and barriers where the first program annotated the graph.

The compiler uses the behavior of each function over its representative input as the behavior of that function over any input, as described in Section 5.2. The compiler must compute the behavior of all mutually recursive procedures together, but in general computes the entries in the function environment in an order determined by a topological sort of the recursive-set nodes in the program. A recursive-set node consists of either a function alone, for non-recursive functions, or a function and all of the functions it calls recursively, for recursive functions. This allows the compiler to compute the function environment for each function f_i before all non-recursive calls to f_i. Computation of input-output mappings for each recursive-set in topological order also speeds up analysis by making the function environment converge faster. The analysis module takes time proportional to the number of recursive-sets and time quadratic in the size of the recursive sets.

In more detail, the first module computes the call graph of the program. From the call graph, the compiler determines the recursive-sets of the program and the order in which function environment entries must be computed. The compiler then generates the representative inputs and computes the function environment in topological order.

Next, the compiler visits each procedure and applies first the deallocation command verification algorithm and then the insertion algorithm. Any time a potentially unsafe deallocation command is found, the compiler issues a warning with as much identification information as possible.

The insertion algorithm works on one control region at a time. Control regions in the program graph correspond to the bodies of procedures, the branches of conditional and case expressions, and the code before a barrier. Each of these regions must have been a `letrec` block in the original KID$^-$ code.

Within each control region, the compiler determines all of the output ports (which correspond to the definition of an identifier in the `letrec` block) that will produce structures whose lifetime is definitely contained by that of the control region. These ports are then sorted by the size of the sets of labels to which they may be bound. Any port whose label set contains another port's label is discarded. This process is repeated until we are left with a set of ports whose label sets are disjoint. The compiler then inserts deallocation commands on each of these ports.

Any time the compiler inserts a deallocation command, it informs the programmer where the deallocation command was inserted. If the components of a structure can be deallocated, the compiler will insert selection and deallocation code for these elements. The compiler has special cases for inserting code to deallocate arrays and their components (shared or unshared) and to deallocate lists recursively (cyclic or acyclic). The current implementation does not insert conditional deallocation commands.

In addition to the two modules that implement the deallocation verification and insertion algorithms, there is a module that generates code to allocate structures in activation frames rather than the heap. This module finds structures of static size that are allocated and deallocated within the same control region and changes them to be frame allocated. Restricting this module to apply only to structures allocated and deallocated within the same control region — rather than within the same procedure — limits its usefulness slightly. Nevertheless, this module is fairly effective at converting general allocation and deallocation code into frame-based allocation and deallocation code. The restriction that the sizes of frame-allocated objects must be known at compile-time is imposed by the Id Run Time System (Id-RTS) [41] on the Monsoon dataflow machine [36] which must know the complete size of an activation frame before a procedure is called. We discuss the effectiveness of this optimization in Section 10.3.

10.1.2 Monsoon

Monsoon [36] is a dataflow machine with an explicit token store. Instead of using a hashing function to match the token pairs associated with instruction instances, each instruction has an explicit address (relative to an activation or frame pointer) where operand matching occurs.

All of the experiments described in this chapter were run on a configuration of Monsoon hardware consisting of one processor and one I-structure unit. Each monsoon processing element (PE) contains 256K 32-bit words of instruction memory, 256K 64-bit words of data memory used for activation frames, and 256K element token queues. The processor consists of an eight stage pipeline operating at 10 MHz. Eight different threads of computation are interleaved in the pipeline.

Each I-structure (IS) unit consists of 4M 64-bit words of data memory. Each word of data memory on both the PE and IS boards has an associated 3 presence-bits and 8 type-bits. The presence bits indicate whether a word of memory is empty or present and are the basic mechanism for fine-grain synchronization on Monsoon. The presence bits in activation frames are used for operand matching while the presence bits in heap memory are used to implement I-structure semantics.

Monsoon is heavily instrumented. Each processor has a statistics processor, containing 64 statistics registers, that counts on a cycle-by-cycle basis what type of operations were executed and to what group of procedures those operations belonged. One of these counters is incremented every cycle. The counters are divided into 8 banks of 8 counters. The counter to be incremented is determined by the operation type and a 3-bit color field from a executing token's continuation. For most operations, the 3-bit color field is used to choose one of the first 7 banks of counters, and the operation type is used to choose one of the 8 counters in the chosen bank. Events such as idle pipeline cycles are counted in the last bank of 8 counters.

These statistics counters allow us to measure the utilization of the machine very precisely. We can account for how much time is spent in the user's program, how much is spent in the Run-Time System (RTS), and how much is spent with the processor idle. We use the statistics counters to measure the performance of our examples.

10.1.3 Id Run-Time System on Monsoon

The version of the run-time system that we used when running these experiments consisted of a frame manager and a heap manager. The frame manager uses a single free list to manage unused activation frames, and so it only allocates one size of activation frame. The run-time system is initialized so that this frame size is large enough for all procedures.

The heap manager uses the quick-fit algorithm [43] to manage deallocated storage. This algorithm incurs one word of overhead for all objects that are allocated. This overhead is insignificant for large objects, but is significant for small objects such as `cons` cells. Under this management strategy, `cons` cells take three words apiece.

All structures in Id are implemented as I-structures. Each word of an I-structure has *presence-bits* that indicate whether that word is *empty* or *present*. Stores cause the presence-bits of a word to go from empty to present as well as changing the value of the word. Fetches issued against an empty word *defer* until a value is stored in that word.

One of the duties of the heap manager is to clear the presence-bits of each word of memory to empty. The current Id RTS clears the presence-bits of all words of the heap during an initialization phase before a program is executed. During program execution, presence-bits are cleared whenever an object is deallocated. The heap manager maintains the invariant that all of the presence-bits of free memory are empty.

In steady-state, when as many objects are being deallocated as are allocated, it does not matter whether presence-bits are cleared upon allocation or upon deallocation. However, if presence-bits are cleared upon deallocation, the difference in run-time between programs that reclaim storage and those that never reclaim storage can be significant — programs that do not reclaim storage are not charged for clearing the presence-bits of the I-structures that they allocate. Under this strategy, a program that does not reclaim storage will have better performance than one that does reclaim storage unless it runs out of memory.

We find that most of our programs that allocate and deallocate approximately equal amounts of storage spend about half their time in the run-time system. Of the time spent executing run-time system code, half is spent clearing presence-bits, and the other half is spent manipulating the data structures that keep track of free and allocated storage.

The activation frame and heap managers both contain code to record the maximum amount of storage that was allocated and the current amount of storage allocated. We use this code to gather statistics about the amount of storage used by our example programs.

10.1.4 Structure of the Experiments

For each program we studied, we determined storage usage and execution time without storage deallocation, and storage usage and execution time with the best hand-inserted deallocation. Then we recompiled the programs to verify the hand-inserted deallocation commands, recording the percent increase in compile-time and the static percentage of deallocation commands verified. We also recompiled the original programs to insert deallocation commands automatically, again recording the percent increase in compile-time and the static percentage of deallocation commands inserted. Finally, we ran the programs again to determine dynamic storage usage and execution time for the programs with verified deallocation commands only and automatically inserted deallocation commands only.

10.2 Performance Measurements

This section describes the compile-time performance of our implementation of the verification and insertion algorithms. It also describes the run-time performance of the various versions of the compiled code. The first example described is the Wavefront benchmark. Wavefront is an example we use to illustrate the use of non-strictness in the definition of relaxation programs. The second example described is the Simple hydrodynamics benchmark. Both Wavefront and Simple are programs with very static structure. Both of these programs use arrays as their major data structure. The third example described is the Gamteb benchmark. This example has a more dynamic structure, because the heart of the simulation is a set of 7 mutually recursive procedures. Gamteb allocates a large number of tuples as it simulates the trajectories of photons in a carbon rod.

```
def multiwave edge_vector n =
  {m = initial_wave edge_vector ;
   r =
     {for i <- 1 to n do
        next m = wave m ;
       finally m }
   in r };
```

Figure 10.1: The code for `multiwave`

```
def multiwave edge_vector n =
  {m = initial_wave edge_vector ;
   r =
     {for i <- 1 to n do
        next m = wave m ;
       ---
        Dealloc(m);
       finally m }
   ---
    _ = if (1 <= n) then Dealloc(m);
   in r };
```

Figure 10.2: The annotated code for `multiwave`

10.2.1 The Wavefront Benchmark

The Wavefront benchmark is a simple example used to test automatic storage reclamation. The outer loop of the example is shown in Figure 10.1. Procedure `initial_wave` allocates a matrix, and each iteration procedure `wave` reads matrix `m` and creates a new matrix. The matrix passed into each iteration of the loop is garbage upon termination of that iteration. The analyzer correctly determines this and allows the compiler to generate the code in Figure 10.2.

We can reclaim the storage associated with the value of `initial_wave` whenever the loop executes at least once.

The following table contains the compile-times for the Wavefront benchmark. The four versions of the program are Wavefront$_{NA}$, Wavefront$_{HA}$, Wavefront$_{VF}$ and Wavefront$_{AA}$. Wavefront$_{NA}$ is the original version, without any deallocation commands. This program was compiled by the unmodified Id compiler. Wavefront$_{HA}$ is a hand-annotated version that contains deallocation commands that were inserted manually. It was also compiled with the unmodified Id compiler. Wavefront$_{VF}$ is the hand-annotated version as compiled by the Id compiler with the lifetime analysis and deallocation verification module. All unsafe deallocation commands are removed by the compiler. Wavefront$_{AA}$ is the unannotated version of the Wavefront program compiled with both the lifetime analysis and deallocation insertion modules. The number of deallocation commands is a static count of all of the deallocation commands in the program.

Program	Compile-Time (seconds)	Deallocs (number)
Wavefront_{NA}	18	0
Wavefront_{HA}	18	3
Wavefront_{VF}	32	2
Wavefront_{AA}	32	2

The hand-annotated version contains three deallocation commands to deallocate the edge-vector, the first matrix, and each intermediate matrix. The compiler-verified and compiler annotated version contain two deallocations: one for the edge vector, and one for the intermediate matrices. The compiler cannot determine that the first matrix will not be returned as the result, so it cannot insert code to deallocate that matrix. A programmer can insert conditionals to prevent error in this case.

The following table describes the run-time performance of the four versions of Wavefront. Each program was run 40 iterations on a 30×30 matrix. The table gives the total run-time for each program, as well as the maximum amount of storage that was allocated, in words, and the final number of words of storage that were still allocated when the programs terminated.

Program	Run-Time (seconds)	Max Storage (words)	Final Storage (words)
Wavefront_{NA}	0.193	37,225	37,225
Wavefront_{HA}	0.349	10,000	907
Wavefront_{VF}	0.336	10,000	1814
Wavefront_{AA}	0.375	10,000	941

The original version of this program runs the fastest, but it also uses the most storage. The hand annotated version takes 81% longer. However, it deallocates all but the final matrix. The main reason the versions containing deallocation code take longer to execute is because the deallocation code must clear the presence-bits of the objects being deallocated.

The compiler-verified and compiler-annotated versions deallocate all but the first and last matrices. Deallocation of the first matrix cannot be verified, because if we execute zero iterations, the first matrix is returned as the result, and the compiler cannot prove that we execute more than zero iterations. We discuss this problem in more detail in Section 10.4.

10.2.2 Simple

Simple, a hydrodynamics benchmark program [13], is a scientific program with very simple control structure. If compiler-directed storage reclamation is going to have any success, it should be able to reclaim every intermediate structure allocated in this program. In fact, our first implementation of the program annotator, which did not handle nested structures, had very good success on Simple. It inserted `Dealloc` statements that deallocated seventy percent (dynamically) of the structures allocated by the program at run-time. Unfortunately, these were tuples that contained numbers of large matrices, and so this was a small fraction — only thirty percent for problem size of ten by ten — of the total storage allocated.

The following table contains the compile-times for the Simple benchmark under four conditions: not annotated (NA), hand annotated (HA), verified safe deallocation commands only (VF) and automatically annotated (AA).

Program	Compile-Time (seconds)	Deallocs (number)
Simple_{NA}	409	0
Simple_{HA}	437	70
Simple_{VF}	863	58
Simple_{AA}	894	58

Compilation of the hand-annotated version of Simple took slightly longer than the original version, while compilation with the lifetime analyzer and deallocation command verifier or inserter turned took twice as long as compilation of the original program.

The twelve deallocation commands $(70 - 58)$ that could not be verified as safe were all potentially unsafe because they deallocated structures that may escape if a loop executed zero iterations. These deallocation commands in version HA are actually safe, because the loop never executes fewer than one iteration.

The following table contains information about the run-time performance of the four versions of Simple. Each version was run twice: once for 20 iterations of a 50×50 matrix, and once for 40 iterations of a 50×50 matrix.

Program	Size	Iters	Time (seconds)	Max Storage (words)	Final Storage (words)
Simple_{NA}	50	20	38.9	1,678,867	1,678,867
Simple_{NA}	50	40	77.0	3,324,447	3,324,447
Simple_{HA}	50	20	51.5	114,147	40,941
Simple_{HA}	50	40	102.6	114,147	40,941
Simple_{VF}	50	20	51.5	114,147	58,609
Simple_{VF}	50	40	102.6	114,147	58,609
Simple_{AA}	50	20	51.6	114,147	58,609
Simple_{AA}	50	40	102.5	114,147	58,609

Each version that contains deallocation commands took about 33% longer to run than the version that had no deallocation commands. However, these each deallocated 93% to 97% of the storage that they allocated. Each of the three versions containing deallocation commands reclaims all of the storage allocated during each iteration. The only difference in the amount of storage that they use is in how much of the storage allocated for initial data structures is eventually reclaimed.

10.2.3 Gamteb

Gamteb [8], a Monte Carlo simulation of photon transport in a graphite rod, is another scientific program on which this system should have good success. The Id version has a slightly more complex structure than the original Fortran: the Id version uses a recursive procedure to simulate particle transport. This recursive procedure is called from a parallel outer loop. Each recursive procedure is called with a new particle and returns a new tuple of counts. The particle tuples passed in can be deallocated upon termination of the recursive call, and the count tuples returned as the result of the recursive call are read and may be deallocated upon termination of each invocation of the outer loop.

A version of Gamteb with hand-inserted deallocation commands contained 38 deallocation commands. The compiler verifies the safety of 37 of these deallocation commands. The compiler fails

```
{ def frame_tuple(x,y) =
    k1{ ft = k2MakeTuple(x,y)
      result = k3g(ft);
    in k4result };

  def g(t) =
    k6{ r = k7Select1(t)
    in k8r };

in k9frame_tuple(68,47) }
```

Figure 10.3: Frame allocated tuple example

to verify one deallocation command that reclaims a structure that may be passed through a zero-tripping loop. The compiler can insert 35 deallocation commands. It fails to insert two deallocation commands that reclaim structures that may be passed through zero-tripping loops.

The following table contains the compilation times for the four versions of Gamteb.

Program	Compile-Time (seconds)	Deallocs (number)
Gamteb$_{NA}$	158	0
Gamteb$_{HA}$	183	36
Gamteb$_{VF}$	976	34
Gamteb$_{AA}$	980	34

The following table contains information about the run-time performance of the four versions of Gamteb.

Program	N	Run-Time (seconds)	Max Storage (words)	Final Storage (words)
Gamteb$_{NA}$	1000	10.9	982,315	982,315
Gamteb$_{NA}$	2000	20.3	1,839,952	1,839,952
Gamteb$_{HA}$	1000	18.0	4710	132
Gamteb$_{HA}$	2000	33.6	5100	132
Gamteb$_{VF}$	1000	17.9	50000	48098
Gamteb$_{VF}$	2000	33.6	90000	89398
Gamteb$_{AA}$	1000	17.7	50000	48098
Gamteb$_{AA}$	2000	33.1	90000	89398

10.3 Transformation to Frame Allocation

When the compiler finds a structure that is allocated and deallocated in the same control region, it can transform the heap allocation into frame allocation. Deallocation of the structure then happens automatically when the procedure exits. In other words, the compiler sets aside enough storage in the activation frame of the procedure to contain the structure. In some implementations, such as the implementation of Id on Monsoon, this is only possible if the structure size is known statically.

```
{ def frame_tuple(x,y) =
    k1{ ft = k2MakeFrameTuple(x,y)
      result = k3g(ft);
     ---
      _ = CleanupFrameTuple(ft);
    in k4result };

  def g(t) =
    k6{ r = k7Select1(t)
    in k8r };

in k9frame_tuple(68,47) }
```

Figure 10.4: Frame allocated tuple example with transformation

In other implementations, where activations frames are stack allocated, the procedure may be able to dynamically allocate space in its activation frame by adjusting its stack pointer.

Procedure `frame_tuple` shown in Figure 10.3 contains a tuple bound to identifier `ft` that may be frame allocated, because the structure allocated by expression k_2 in procedure `frame_tuple` does not escape from the invocation of `frame_tuple`.

Figure 10.4 contains the transformed code for this example. The primitive `MakeFrameTuple` allocates a tuple in the frame. The semantics of the tuple is exactly the same as for a heap-allocated tuple, except that the storage is automatically reclaimed upon termination of the procedure `frame_tuple`. The primitive `CleanupFrameTuple` performs any cleanup required by the run-time system. The Id run-time system requires that all frames be empty when returned, and so `CleanupFrameTuple` clears out the storage used by the tuple.

The following table summarizes the results when we compile the hand annotated version of Gamteb with the frame allocation optimization enabled:

Program	Compile-Time (seconds)	Deallocs (number)
Gamteb$_{HA}$	183	38
Gamteb$_{HAFA}$	183	38

The following table contains information about the run-time performance of the Gamteb benchmark compiled with the frame allocation optimization enabled.

Program	N	Run-Time (seconds)	Max Storage (words)	Final Storage (words)
Gamteb$_{HA}$	1000	18.0	4710	132
Gamteb$_{HAFA}$	1000	15.7	3510	132
Gamteb$_{HAFA}$	2000	29.4	3500	132

The version of Gamteb that uses frame allocation runs 13% faster than the original version, and uses less total storage. The optimization itself is very straightforward and does not increase compile-time noticeably.

10.4 Handling Possibly Zero-Tripping Loops

A common idiom in functional implementations of scientific programs is a structure that is created and then successively refined in a loop or tail recursion. Often, only the final value is needed, and the initial value and all intermediate values can be reclaimed. However, if the compiler cannot determine that the loop will execute at least once, then it cannot tell that the final value could not be the initial value, and the initial value will never be reclaimed by the compiler.

Here is such an example:

```
def multiwave ev k =
  { M = initial_wave ev;
   in {for i <- 1 to k do
         next M = wave M;
       finally M }};
```

The initial value of `M`, allocated by `initial_wave`, will be returned as the final value of the loop if the value of `k` is less than one.

We can provide run-time checking to ensure that the initial matrix is only deallocated if it is not returned as the result by testing the initial value of the loop predicate. The following code has this transformation.

```
def multiwave ev k =
  { M = initial_wave ev;
     r = {for i <- 1 to k do
         next M = wave M;
       finally M }
    ---
     _ = if k > 0 then deallocate M;
    in r };
```

The code after the barrier deallocates the initial copy of `M` if `k` is at least one. In Wavefront, this optimization only reclaims one object, so it is not very interesting. We applied the same optimization with much more spectacular results.

The following table summarizes the performance of Gamteb when it is compiled with the zero-tripping optimization turned on. The compile-time for the row labeled Gamteb$_{ZT}$ includes the time to perform the zero-tripping optimization. The compile-time of Gamteb$_{ZTFA}$ includes both the zero-tripping detection and frame-allocation optimizations.

Program	Compile-Time (seconds)	Deallocs (number)
Gamteb$_{AA}$	980	34
Gamteb$_{ZT}$	980	36
Gamteb$_{ZTFA}$	981	36

This optimization takes very little time, but allows the compiler to add two more deallocation commands to Gamteb than it could without the optimization. These two deallocation commands, as we can see from the following table, reduce the storage used by Gamteb considerably. Furthermore, once these two deallocation commands have been added, Gamteb uses a constant amount of storage for any number of particles simulated.

Program	N	Run-Time (seconds)	Max Storage (words)	Final Storage (words)
Gamteb_{HA}	1000	18.0	4700	132
Gamteb_{AA}	1000	17.7	50000	48098
Gamteb_{AA}	2000	33.1	91100	89398
Gamteb_{ZT}	1000	18.0	4900	132
Gamteb_{ZT}	2000	33.6	4900	132
Gamteb_{ZTFA}	1000	15.6	3500	132
Gamteb_{ZTFA}	2000	29.3	3700	132

These performance results show that the zero-tripping loop optimization is very important, even though it only inserts code to deallocate one structure per loop.

We did similar experiments with Simple to see what difference it made to reclaim the storage from structures that may be returned as the result of a loop. The following table shows the compile times and the number of deallocations inserted. The *ZT* version of Simple is compiled with the ZT transformation, which inserts twelve additional deallocation commands.

Program	Compile-Time (seconds)	Deallocs (number)
Simple_{AA}	1100	58
Simple_{ZT}	1000	70

The following table summarizes the results of running the HA, AA, and ZT versions of Simple on a 50×50 problem size for 20 and 40 iterations.

Program	Size	Iters	Time (seconds)	Max Storage (words)	Final Storage (words)
Simple_{HA}	50	20	51.5	114,147	40,941
Simple_{HA}	50	40	102.6	114,147	40,941
Simple_{AA}	50	20	51.6	114,147	58,609
Simple_{AA}	50	40	102.5	114,147	58,609
Simple_{ZT}	50	20	51.6	114,147	40,941
Simple_{ZT}	50	40	102.5	114,147	40,941

Use of the ZT transformation allows the compiler-generated deallocation commands to reclaim as much storage as the hand-generated deallocation commands do.

10.5 Examples Using Lists

This section describes the experiments we did with list manipulating programs. The first example, shown below, creates a list named $l1$ containing len integers. It then creates a list named $l2$ by incrementing each element in $l1$ by $n1$. It creates another list $l3$ by scaling each element in $l2$ by $n2$. Finally, it returns the sum of the elements of list $l3$.

```
def test len n1 n2 =
  { l1 = gen_list len;
    l2 = inc_list n1 l1;
```

```
     l3 = scale_list n2 l2;
     r = sum_list l3;
    in r };
```

The three list generating procedure `gen_list`, `inc_list`, and `scale_list` were written using list comprehensions. In Id, a list comprehension is syntactic sugar that expands into a loop expression that generates a list. List comprehensions tend to make list manipulating programs more compact.

The Id compiler inserts code that allocates one extra `cons` cell for each list comprehension. The extra cell simplifies the code that constructs the list, because it eliminates the extra testing that would be needed otherwise when generating an empty list. The lifetime of the extra `cons` cell is always bounded by the control region enclosing the list comprehension, but the standard Id compiler does not currently insert deallocation code for this extra cell.

The following table shows the compile-time performance of three versions of this program: no annotations inserted (NA), hand inserted deallocation commands (HA), and automatically annotated (AA). The compiler could not verify any of the hand inserted deallocation commands because they are contained in procedures and violate the safety condition that we defined in Chapter 5. The compiler has special cases for inserting code to deallocate lists, and these were used to generate the automatically annotated version of the benchmark.

Program	Compile-Time (seconds)	# Dealloc	# Deallocate_List
List$_{NA}$	11	0	0
List$_{HA}$	15	0	3
List$_{AA}$	26	3	3

The hand annotated version of the List benchmark contains three calls to the procedure `Deallocate_List`, which deallocates all cells of a list. This procedure assumes that the list is acyclic. The hand annotated version does not deallocate the extra `cons` cells allocated by the list comprehension code because there is no way to name these cells in the Id source code. The compiler annotated version of the List benchmark contains three `Dealloc` commands to reclaim extra `cons` cells allocated by the list comprehension code, as well as three calls to `Deallocate_Cyclic_List`, which deallocates all unique cells in a list. The compiler cannot determine that a list is acyclic, and so it inserts code that safely deallocates both cyclic and acyclic lists.

The following table contains information about the run-time performance of the three versions of the list manipulating benchmark.

Program	Length	Run-Time (seconds)	Max Storage (words)	Final Storage (words)
List$_{NA}$	1000	0.194	9009	9009
List$_{NA}$	100,000	19.3	900,009	900,009
List$_{DA}$	1000	0.426	9009	9
List$_{DA}$	100,000	42.5	900,009	9
List$_{AA}$	1000	0.498	9006	0
List$_{AA}$	100,000	49.3	900,006	0

Both versions of this benchmark that deallocate storage take more than twice as long as the original code. The compiler annotated version of this benchmark uses the least amount of storage, but takes the longest because the code to deallocate a potentially cyclic list is more expensive than the code to deallocate an acyclic list. The automatically annotated version has a lower maximum storage because the deallocation of one of the extra `cons` cells was allocated before the both of the others were allocated.

10.6 Explicit Storage Reuse

If the compiler finds a structure that is allocated in each iteration of a loop and deallocated in the following iteration, then the compiler can lift both the allocate and the deallocate out of the loop and explicitly reuse the structure. In some cases the compiler may have to allocate two or more structures outside of the loop and cycle through them.

Consider the following example, where `M` is a matrix that is successively relaxed. In each iteration, a new version of `M` is created and an old one becomes garbage. Furthermore, the loop is *bounded* by parameter `k` — this allows up to `k` iterations of the loop to execute in parallel. Therefore, the space used by the loop should be bounded by `k` times the space requirements of a single iteration.

```
def relax M size n_steps =
  {for i <- 1 to n_steps bound k do
     next M = {matrix (1,size),(1,size) of
               | [i,j] = relax_point M i j
                 || i <- 1 to size & j <- 1 to size };
    ---
     Dealloc(M)
    finally M };
```

Although this version of the procedure reclaims all intermediate storage allocated, it calls the heap manager `n_steps` times to allocate storage and `n_steps` times to deallocate storage. We only ever need `k` instances of the matrix `M` at any point in time, and so we should be able to locally manage the storage in order to reduce the burden on the heap manager. We would like to specialize storage management whenever possible to increase the efficiency for particular uses of storage.

The previous procedure definition can be transformed into the following code in order to reduce the overhead of storage management.

```
def relax M size n_steps =
  { Ms = make_k_matrices ((1,size),(1,size)) k;
    R = {for i <- 1 to n_steps bound k do
           next M = Ms![i mod k];
           _ = {for i <- 1 to size do
                  {for j <- 1 to size do
                     next M[i,j] = relax_point M i j }};
          ---
           Ms![i mod k] = clear_matrix M;
         finally M };
    ---
     _ = free_k_matrices Ms;
    in R };
```

The procedure `make_k_matrices` takes the dimensions `b` of the matrix and the loop bound `k` and returns an M-vector [7, 6] containing `k` empty matrices each with dimensions `b`. Each iteration, the (`i` mod `k`)th element of the vector of empty matrices is `taken` and used as the value of `next M`. Upon termination of the `i`th iteration, the current value of `M` is cleared and *put* back into the (`i` mod `k`)th of the vector of empty matrices. The vector of empty matrices and all of the empty matrices are deallocated upon termination of the whole loop by the call to `free_k_matrices`.

This optimization is not currently implemented, but we expect it to be effective in reducing the run-time overhead of allocating and deallocating storage.

Chapter 11

Conclusion

We have presented a method for performing object lifetime analysis on non-strict, parallel programs. We have shown how to use this lifetime information to verify the correctness of deallocation commands in programs and to insert deallocation commands into programs. The central idea of this work is recognizing that object lifetimes can be derived from reachability information, and that interpreters can determine what objects are reachable from any point in the program.

The crux of the analysis is the naming of objects. Object names must be related to program structure so that dynamic behavior can be related to the static structure of a program. Once we have realized that, it is straightforward to derive an abstract interpretation that yields a summarization of object reachability. We have presented an operational semantics that derives object names from the dynamic structure of a program's call tree. We discussed several abstractions of this naming scheme that allow us to model the allocation and connectivity of objects with varying degrees of precision.

The technique of using abstract interpretation to derive an analysis method from the semantics of a programming language shows great promise. The lifetime analysis presented in this thesis is precise enough to yield great reductions in the usage of storage in many non-trivial scientific applications. Our experiments showed that deallocation code inserted by the compiler could reclaim eighty to one hundred percent of the storage allocated by a program. While we do not claim that compilers will have this level of effectiveness for all programs, we do claim that there is a large class of programs for which these methods are very effective.

11.1 Further Research

This thesis is by no means the last word in lifetime analysis. We have taken another step by defining a lifetime analysis framework for non-strict, parallel languages, but a number of issues remain to be investigated.

11.1.1 Computing Object Lifetimes

The algorithm that we described to compute function environments in the abstract interpreter is very straightforward, but not necessarily very efficient. The process of computing function environments needs to be as efficient as possible if abstract interpretation is going to be a practical tool.

11.1.2 Subscript Analysis

The interaction of subscript analysis and abstract interpretation is an area that could be explored further. Can we do better analysis of programs using arrays if we can determine that certain arrays have distinct subregions with potentially different behaviors? For instance, in some scientific programs, arrays are created where all of the border elements are shared and all of the inner elements are unique. If we could use subscript analysis to distinguish these regions during abstract interpretation, we might be able to determine that all of the interior elements could be deallocated without having to test for uniqueness.

11.1.3 Determining Acyclicity of Recursive Objects

We feel that, by modifying our abstract interpreter, we should be able to perform sharing analysis of recursively-typed objects. The goal of this sharing analysis would be to annotate recursively typed object representations to indicate whether they form trees, acyclic graphs, or cyclic graphs. This information should allow us to distinguish between objects that are definitely trees, objects that are definitely acyclic and objects that may be cyclic. This information would be useful because the compiler can generate more efficient code to reclaim trees and lists than to reclaim graphs and cyclic structures.

Hendren [20] and Harrison [19] both can determine whether objects are acyclic using information about the allocation time of the nodes of a recursively typed object. They used this information to determine when statements or subexpressions could be executed in parallel. However, their methods depend on having a sequential interpreter, so the methods do not apply to our work.

The insight we had that allowed us to collect sharing information for the elements of arrays created with `MakeArray` should carry over to recursive objects: the `MakeArray` construct provides a good encapsulation of the expression evaluated to obtain the elements of an array. We can determine if sharing is possible by observing the boundary of the encapsulation and seeing if any objects cross it, or are inherited. The values that cross the boundary may be shared by the different elements of the array.

Basically, we need to unfold a recursive function once during analysis to determine if the recursive calls to the function can share values with the initial call to the function. If there is no sharing between the initial call and the recursive calls, then there can be no sharing between any of the calls because each of the recursive calls can be considered to be an initial call. Unfortunately, we have not seen how to formalize this condition in such a way that it can be included in our lifetime analysis method. If we proceed to unfold every recursive call once during abstract interpretation, then abstract interpretation will not terminate. Every iteration of the computation of the function environment will yield one more input value to which the recursive function must be applied.

Lent[30] explored the selective unfolding of recursive procedure calls to determine acyclicity of lists. He proposed a special mechanism for unfolding function calls one extra time using renamed labels and then collapsing the renamed values back into the original domain. This extra level of labels should allow us to detect sharing and to annotate the unshared objects, so that we can preserve the sharing information once the labels are recompressed. We would like to investigate this technique in more detail to determine if it is sound and to extend it beyond detecting acyclic lists to detecting tress or directed acyclic graphs.

11.1.4 Deallocating Complex Structures

The problem of generating code to deallocate complex structures is related to the problem of determining the acyclicity and sharing of complex structures. The current implementation of the deallocation insertion algorithm in the Id compiler has a few special cases for inserting code to deallocate single `cons` cells, potentially cyclic lists, and acyclic lists. The problem of generating code to traverse and deallocate recursive objects is still open. The compiler may be able to generate a procedure for each type to deallocate complete objects of that type. The compiler could then compose these special deallocation procedures to deallocate objects consisting of the composition of several types of objects.

The problem of deallocating nested or recursive structures is exacerbated when the pattern of sharing within the structure is complex or unknown. Perhaps the run time system could provide a function that recursively descends a structure and deallocates all unique objects in that structure.

11.1.5 Interaction with Garbage Collection

Another area that deserves more attention is the interaction of explicit storage management with garbage collection. Is it really possible for the two to coexist such that the use of explicit deallocation commands decreases the overhead of garbage collection? One approach that we think is worth considering is having the compiler generate code to allocate storage in an area separate from the garbage collected heap. This code can explicitly deallocate the whole area when the objects in it are all dead.

Another possibility is to have a dynamic storage manager and a garbage collector that coexist in one space. Explicit deallocation commands can be used to deallocate storage. Whatever storage is not deallocated explicitly will eventually be deallocated by the garbage collector.

11.1.6 M-Structures

Full-fledged Id and KID both have M-structures [7, 6], which are useful when writing programs that compute histograms, implement graph algorithms, or implement run-time system code. M-structures are mutable structures that allow mutually exclusive access to each word.

We would like to see our instrumented and abstract interpreters augmented to handle programs using M-structures. We believe that M-structures can be modeled safely in our abstract interpreter in the same fashion as I-Structures. But our solution for modeling abstract M-structures does not solve the problem of modeling M-structures in the instrumented interpreter. It seems that the store would have to be threaded through the interpreter in order for the interpreter to model mutually exclusive access to each M-structure element. We would like to find a solution to this problem that does not obscure the parallelism of the interpreter.

Once M-Structures are added to KID$^-$ we will have to model barriers in full generality, which may involve computing a graph of activation label precedence. This precedence relation would be analogous to our *terminates before* relation. Once we have computed the precedence graph, we may be able to determine in some cases whether programs deadlock.

11.2 Other Research Directions

Other semantic analyses are useful for a wide variety of reasons. Strictness analysis is helpful in determining that portions of a program may be sequentialized. Sequentialization is a useful optimization for compiling non-strict languages because it eliminates redundant synchronization. Dependence analysis and interference analysis are also important analyses in the field of ptimizing compilers..

The abstract interpretation framework presented in this thesis is a sound basis for a wide variety of other such analyses of non-strict or parallel programs. By changing the abstract evaluators and value domains presented in this report, the abstract interpreter can be restructured to support these other data dependent analysis methods.

Bibliography

[1] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the 18th ACM Symposium on the Principles of Programming Languages*, pages 279–290, New York, NY, January 1991. Association for Computing Machinery, ACM Press.

[2] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25:275–279, 1987.

[3] Zena M. Ariola and Arvind. Compilation of Id-: a subset of Id. Computation Structures Group Memo 315-1, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, December 1990.

[4] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *1990 ACM Conference on Lisp and Functional Programming*, pages 218–226, New York, NY, June 1990. Association for Computing Machinery, ACM Press.

[5] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[6] Paul S. Barth. *Atomic Data Structures for an Implicitly Parallel Language.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, January 1992.

[7] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, pages 538–568, Berlin, August 1991. Association for Computing Machinery, Springer-Verlag.

[8] Patrick J. Burns, Mark Christon, Roland Schweitzer, Olaf M. Lubeck, Harvey J. Wasserman, Margaret L. Simmons, and Daniel V. Pryor. Vectorization of Monte Carlo particle transport: An architectural study using the LANL benchmark "Gamteb". In *Proceedings Supercomputing '89*, pages 10–20, New York, NY, November 1989. IEEE Computer Society and ACM SIGARCH, ACM Press.

[9] David C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, May 1989. (Technical Report CS-89-108).

[10] David R. Chase, Mark Wegmen, and F. Kenneth Zadeck. Analysis of pointers and structures. In *SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, New York, NY, June 1990. SIGPLAN, Association for Computing Machinery.

[11] Douglas W.. Clark. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–86, February 1977.

[12] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on the Principles of Programming Languages.* Association for Computing Machinery, 1977.

[13] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. UCID 17715, Lawrence Livermore Laboratory, February 1978.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *Conference Record of the 16th ACM Symposium on the Principles of Programming Languages*, pages 25–35, January 1989.

[15] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the 17th ACM Symposium on the Principles of Programming Languages*, pages 157–168. ACM SIGACT-SIGPLAN, January 1990.

[16] McGraw et al. SISAL: Streams and iteration in a single-assignment language. Report M-146, Rev. 1, University of California, Lawrence Livermore National Laboratory, Livermore, California, March 1985.

[17] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 Reference Manual. LCS-TR 407, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1987.

[18] David K. Gifford, Pierre Jouvelot, and Mark A. Sheldon. Report on the FX-90 programming language. Programming systems research group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1990. Draft.

[19] Williams Ludwell Harrison, III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[20] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Computing*, 1(1), January 1990.

[21] Harry S. Hochheiser. A Schezoid Compiler for P-RISC. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[22] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, New York, NY, June 1989. ACM Press.

[23] Paul Hudak. A semantic model of reference counting and its abstraction. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, Computers and Their Applications, chapter 3, pages 45–62. Ellis Horwood Limited, Chichester, West Sussex, England, 1987.

[24] Paul Hudak and Jonathan Young. A collecting interpretation of expressions (without powerdomains). *Conference Record of the 15th ACM Symposium on the Principles of Programming Languages*, pages 107–118, January 1988. (Full version accepted to TOPLAS).

[25] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 199–203. Springer-Verlag, Berlin, September 1985.

[26] Thomas Johnsson. Analysing heap contents in a graph reduction intermediate language. In *Proceedings of the 1990 Glasgow Workshop*. Springer-Verlag, 1990.

[27] Neil D. Jones and Steven S. Muchnik. A flexible approach to interprocedural data flow analysis and programs with recursive data structure. In *Conference Record of the 9th ACM Symposium on the Principles of Programming Languages*, pages 66–74, New York, NY, 1982. Association for Computing Machinery, ACM Press.

[28] Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *FPCA Conference Proceedings Fourth International Conference*, pages 54–74, New York, NY, 1989. ACM IFIP, ACM Press.

[29] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, New York, NY, July 1988. SIGPLAN, Association for Computing Machinery.

[30] Arthur Lent. Compile-time analysis of list sharing using abstract interpretation. 6.847 term paper, Massachusetts Institute of Technology, Cambridge, MA, December 1991.

[31] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the 15th ACM Symposium on the Principles of Programming Languages*, pages 47–57. SIGPLAN, ACM Press, January 1988.

[32] David A. Moon. Garbage collection in a large lisp system. In *1984 ACM Conference on Lisp and Functional Programming*, pages 235–246, New York, NY, August 1984. Association for Computing Machinery, ACM Press.

[33] R. S. Nikhil. Id Nouveau Reference Manual Part I: Syntax. Technical report, Computation Structures Group, MIT, 545 Technology Square, Cambridge, Massachusetts, April 1987.

[34] R. S. Nikhil, K. Pingali, and Arvind. Id Nouveau. Computation Structures Group Memo 265, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 1986.

[35] Rishiyur S. Nikhil. Id 90 reference manual. Computation Structures Group Memo 284-1, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, MA, September 1990.

[36] Gregory M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1988.

[37] Young Gil Park and Benjamin Goldberg. Escape analysis on lists: Optimizing storage allocation and reclamation in higher order functional languages. Extended Abstract, 1990.

[38] John E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California, Davis, Livermore, California, 94550, November 1987. (Technical Report UCRL-53832).

[39] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the 15th ACM Symposium on the Principles of Programming Languages*. ACM SIGACT-SIGPLAN, January 1988.

[40] Kenneth R. Traub. A compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1986.

[41] Kenneth R. Traub, Michael J. Beckerle, James E. Hicks, Gregory M. Papadopoulos, Andrew Shaw, and Jonathan Young. Monsoon Software Interface Specifications. Technical Report MCRC-TR-1 and CSG Memo 296, Motorola Cambridge Research Center and Massachusetts Institute of Technology, Cambridge, MA, January 1990.

[42] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, Computers and Their Applications, chapter 12, pages 266–275. Ellis Horwood Limited, Chichester, West Sussex, England, 1987.

[43] C. B. Weinstock and W. A. Wulf. Quick Fit: An Efficient Algorithm for Heap Storage Allocation. *SIGPLAN Notices*, 23(10):141–148, 1988.

[44] Jonathan Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages*. PhD thesis, Yale University, May 1989.

[45] Jonathan Young and Patrick O'Keefe. Experience with a type evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581. North Holland, 1988.

Biographical Note

James Hicks was born on December 28, 1964 in Los Angeles, California. He attended the Massachusetts Institute of Technology. He married Sharon Chang in 1987. He received his S.M. in Computer Science and Engineering in June, 1988 as a VI-A cooperative student working at MIT Lincoln Laboratory. He did his graduate work at MIT in the Computation Structures Group, studying parallel languages, parallel architectures, compilers, and semantic analysis. He completed his Ph.D. in Electrical Engineering and Computer Science in January, 1992. He is a member of Eta Kappa Nu, the national electrical engineering honor society, and the Association for Computing Machinery.