A NEW PERSPECTIVE ON PARTIAL EVALUATION AND USE ANALYSIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Morris J. Katz
January 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Mitchell
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Daniel Weise

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Carolyn Talcott

Approved for the University Committee on Graduate Studies:

Abstract

Partial evaluators are compile time optimizers achieving performance improvements through a program modification technique called *specialization.* Partial evaluators produce one or more copies, or *specializations*, of each procedure in a source program in the output program. Specializations are distinguished by being optimized for invocation from call sites with different characteristics, for example, placing certain constraints on argument values. Specializations are created by partially executing procedures, leaving only unexecutable portions as *residual code. Symbolic execution* can replace variable references by the referenced values, executed primitives by their computed results, and function applications by the bodies of the applied functions, yielding inlining.

One core challenge of partial evaluation is selecting what specializations to create. Attempting to produce an infinite number of specializations results in divergence. The *termination mechanism* of a partial evaluator decides whether or not to symbolically execute a procedure in order to create a new specialization.

Creating a termination mechanism that precludes divergence is not difficult. However, crafting a termination mechanism resulting in the production of a sufficient number of appropriate specializations to produce high quality residual code while still terminating all, or most, of the time is quite challenging. This dissertation presents a new type of analysis, called *use analysis*, forming the basis of a termination mechanism designed to yield a better combination of residual code quality and frequent termination than the current state-of-the-art.

Most termination mechanisms characterize applications based on the arguments supplied to functions during symbolic execution. Possible combinations of arguments to each function are partitioned into sets. At most one specialization of each function is created for each partition.

Use analysis differs by defining partitions based only on information utilized in performing computations during symbolic execution that contributes to the result of a program. Irrelevant information supplied in arguments does not effect termination decisions so a better combination of residual code quality and termination results.

The current implementation of use analysis consumes too many computational resources to be of practical use. Future research is needed to investigate more efficient techniques for implementing use analysis or a more efficient termination mechanism yielding similar results to use analysis.

Acknowledgments

I would like to thank Daniel Weise for creation of the Fuse research group within which this research was performed. While discussions with many members of the group over the years influenced my work, special thanks are due to Erik Ruf whose insights into partial evaluation have been a constant source of ideas and inspiration. Without Erik's assistance I would never have gotten over the many stumbling blocks and avoided the pot holes on the road to completion of this work.

The clarity of my written dissertation owes much to the comments of many reviewers. I would like to thank the members of my reading committee for their thorough reading of numerous revisions, their didactic comments, and their ongoing advice. Particular thanks are due to those individuals who agreed to comment on drafts of the dissertation despite their having no obligation to do so. Thank you Neil Jones, Olivier Danvy, and Erik Ruf.

Finally, I would like to thank those individuals and organizations who helped support my graduate school studies: the Stanford Department of Computer Science for three quarters of support, John Koza for three quarters of support, Microsoft Corporation for one quarter of support, Daniel Weise for one quarter of support, and Scott Seligman for one quarter of support.

Contents

List of Figures

Chapter 1

Introduction

I define a partial evaluator as an optimizer that achieves performance improvements through the utilization of a program modification technique called *procedure cloning* [15]. For each procedure in a source program, the partial evaluator produces one or more copies of the procedure in the output program. When multiple copies appear in the output, or *residual program*, the *clones* are distinguished by their being optimized for invocation from a subset of the call sites in the residual program. Optimizations are performed on the body of that clone that are safe (i.e., correctness preserving) only when the clone is called from that subset. The same optimizations might not be safe in general (i.e., for all call sites).

In the partial evaluation literature, optimized clones are called *specializations*. They are specialized in the sense that each clone can only be correctly utilized for a limited number of call sites that possess certain characteristics. For example, one clone of the exponentiation procedure in Figure 1.1 might be specialized for an exponent of `1`. This specialization can be implemented very efficiently, as shown in Figure 1.2. The specialization needs only return its second argument. This clone can only be called from call sites where the exponent argument is known always to be `1`.

Creating clones typically creates new procedure call sites. These new call sites offer the opportunity to produce new clones. The termination issue in partial evaluation is guaranteeing that only a finite number of clones are investigated and created. Assuring termination when processing recursive source programs is of particular interest since an infinite unrolling of a recursion will take place if a distinct clone is

```
(define expon
  (lambda (base exp)
    (cond
     ((zero? exp) 1)
     ((= exp 1) base)
     ((negative? exp) (/ 1 (expon base (- exp))))
     (#t (* base (expon base (-1+ exp)))))))
```

Figure 1.1: Exponentiation function

```
(define expon
  (lambda (base exp)
    base))
```

Figure 1.2: A specialization of the exponentiation function for an exponent of `1`

created for each iteration of the recursion.

Termination is tricky because the termination mechanism acts as an arbiter amongst several conflicting goals. A partial evaluator must terminate on a sufficiently broad class of input programs to be a useful tool. However, it must produce enough appropriate specializations to yield residual programs that are sufficiently more efficient than the original source. Furthermore, a partial evaluator must not produce too many inappropriate specializations or the bloated residual program might end up slower than the original source.

A partial evaluator must make two intimately related decisions: what clones to create and which clone to call from each call site. There is an additional important decision that is made by all partial evaluators: when should clones be inlined? Most partial evaluators perform extensive inlining of clones. Many inline all procedure calls except those retained to prevent infinite inlining of recursions [30, 37].

Most partial evaluators immediately decide whether to create a new clone for a given call site, which clone to call from the call site, and whether the selected clone will be inlined [30, 37]. Ensuring termination through the selection of a finite number of specializations is complicated in these partial evaluators by the unified approach utilized in making all three decisions. Both Osgood [32] and I have attempted to simplify and improve the decision making process by partitioning partial evaluation into two phases, yielding what I call *delayed commitment partial evaluation.* In the first phase, which I call the *analysis phase*, the partial evaluator investigates *potential specializations* that might, or might not, appear in the final residual program. In the second phase, which I call the *code generation phase*, the partial evaluator selects those potential specializations that will be included in the residual program and makes all the inlining decisions.

Delayed commitment partial evaluation offers leverage on the dual problems of termination and residual code quality. Since the analysis phase only investigates potential specializations, the termination mechanism for the analysis phase only seeks to insure that a finite number of potential specializations are analyzed and that all interesting specializations are analyzed. It does not have to worry about producing too many potential specializations (except as that impacts the run time and resource consumption of the partial evaluator). The code generation phase performs the roles of winnowing out the potential specializations, choosing a specialization to call from each call site, and deciding whether to inline.

As Weise and I described in [31], it is not possible for a partial evaluator both to guarantee termination on all input programs and to produce residual code that manifests a number of desirable properties. In particular, a requirement that a residual program perform no computations at runtime that could have been performed during partial evaluation mandates that the partial evaluator be allowed to diverge on some input programs. Consequently, I propose that a good termination mechanism is one maximizing the cases in which a partial evaluator terminates while still allowing for aggressive optimization.

The purpose of any optimization technique is to improve performance while maintaining unmodified the input/output behavior of a program. In the case of partial evaluation, this means the residual program must produce the same results as the source program for all inputs adhering to the input specification. By implication, the specializations applied at individual call sites must work in concert to preserve this invariance.

Specializations of a function might be characterized by the domain of argument values for which the specialized function has the same input/output properties as the original function. However, requiring specializations to produce the identical result to the specialized source function for all argument values supplied at a call site limits the application sites for each specialization. In some cases, not all of the information returned by a function is utilized in subsequent computations. Any specialization returning a value identical to the source function in those aspects of the return value that are utilized is correctness preserving. This concept naturally yields a parametric characterization of each specialization. Given some constraints on the aspects of the results of a function that must remain unchanged, the domain of values for which a specialized function produces the same result as a source function can be specified.

I propose that not only restrictions on the values supplied as arguments at various call sites, but also restricted utilization of the results of functions, ought to form the basis of selection of the specialization to be utilized at each call site. I further suggest that this same characterization ought to form the basis of the termination mechanism for partial evaluation. Since determining whether two functions are equivalent is undecidable, I have devised a system for approximating the desired characterization by looking at the information utilized about argument values in performing the optimizations used in creating specializations. In my partial evaluator, as well as all others of which I am aware, the sole optimization technique utilized is symbolic execution of code during partial evaluation followed by replacement of the symbolically evaluated expressions by the results computed during partial evaluation.

I propose that the only important information and computations in any program are those utilized in computing the result of the program. Any calculations not causally contributing to the computation of the result of a program are irrelevant, as elimination of those computations would not effect the input/output behavior of the program. By extension, computations performed by the specialized function at runtime in order to compute those portions of its result causally contributing to the result of the program are significant. Since the computations performed at runtime by a specialization are those not performed as part of creating the specialization during partial evaluation, specializations can be characterized by the information used in optimization of computations performed by the source function causally contributing to the result of the program.

The job of all termination mechanisms is to guarantee that only a finite number of specializations are created. While the approaches to termination differ widely, termination can always be reduced to a question of whether a new specialization will be equivalent in some manner to a specialization that has already been, or is being, produced. I propose that the basis of specialization equivalence ought to be the information utilized about argument values used in performing optimizations of computations in the source function causally contributing to computation of the result of a program.

The termination strategy presented herein is based on a new form of analysis that I call *eager use analysis*, and an improved version of use analysis called *lazy use analysis*. These analyses not only are useful in making termination decisions in a partial evaluator, but they can also be used in making code generation decisions that improve the quality of residual code. Furthermore, the analysis techniques used in selecting potential specializations may also have applications to other areas of research beyond partial evaluation. For example, many compiler optimization algorithms are based on source program analyses whose accuracies are limited by the selection of the number of variants of each source procedure for which information is collected. In particular, lazy use analysis offers a new means of selecting both the number and the character of those variants and thereby offers the possibility of significantly improved accuracy.

A partial evaluator utilizing lazy use analysis as its termination mechanism must have a somewhat different structure than traditional partial evaluators. Whereas the termination mechanisms of most partial evaluators indicate when symbolic execution of recursions ought to be terminated due to a detected equivalence, lazy use analysis indicates when symbolic execution ought to be continued because of detected differences. The distinction is what the termination mechanism does in those cases when it is unable to decide whether two applications are equivalent. A partial evaluator utilizing lazy use analysis always terminates recursions unless the analysis indicates that symbolic execution ought to be pursued further. In this regard, lazy use analysis might better be called a *commencement* mechanism.

The very different approach to termination taken by partial evaluators utilizing lazy use analysis leads in some cases to *premature termination*. Premature termination is when basic lazy use analysis fails to cause symbolic execution of a recursion to proceed until all desirable potential specializations are investigated. In order to address the problem of premature termination, an additional analysis, called *base case analysis*, is added to lazy use analysis.

Base case analysis ensures that symbolic execution of all recursions proceeds until at least one base case of the recursion is investigated. The addition of base case analysis to lazy use analysis is motivated by the observation that no recursion can terminate at runtime until a base case is executed. The addition of base case analysis to lazy use analysis appears to significantly mitigate the problem of premature termination.

The remainder of this document is divided into 5 chapters. Chapter 2 presents a conceptual description of termination based on use analysis. Chapter 3 delves into significantly greater detail regarding the utilization of use analysis as a termination mechanism for the analysis phase of a partial evaluator. It also presents the low level details of a sample implementation. Chapter 4 places this work within the context of previous partial evaluation research. It discusses traditional approaches to termination of partial evaluation, their limitations, and how this work differs. Chapter 5 discusses the sources of heavy resource consumption, notably memory, of my termination mechanism. Implementation optimizations that have been employed, both successfully and unsuccessfully are presented. Finally, untested ideas for reduced resource consumption are given. Conclusions and future work appear in Chapter 6.

Chapter 2

Termination and Use Analysis

This chapter presents lazy use analysis as a potential solution to the termination problem in partial evaluation. The chapter begins by explaining the relationship between termination and investigation of a finite number of potential specializations. Equivalence classes are then described as a means of ensuring finiteness and the types of equivalence classes utilized by several different partial evaluators are discussed. Next, the concepts behind use analysis are presented and an algorithm for implementing eager use analysis is described. After a brief discussion of the shortcomings of eager use analysis, lazy use analysis is presented. The chapter concludes with an explanation of the sources of approximation inherent in lazy use analysis and a description of an addition to lazy use analysis called base case analysis, designed to minimize the effects of approximation.

2.1 Finite Potential Specializations

The key to termination of the analysis phase of a partial evaluator is guaranteeing that only a finite number of potential specializations are investigated. It is the responsibility of the termination mechanism to make the selection of the finite set of potential specializations for each input program and input specification. While only finiteness is required for termination, the choice of the set of potential specializations to be investigated can greatly impact the performance of the resulting residual program.

```
(define polymorphic-double
  (lambda (arg)
    (cond
     ((number? arg)
      (+ arg arg))
     ((string? arg)
      (string-append arg arg))
     ((list? arg)
      (append arg arg))
     (else (error "Bad polymorphic-double:" arg)))))
```

Figure 2.1: A polymorphic doubling function that operates on numbers, strings, and lists

Figure 2.1 contains a very simple example of a function for which a possibly infinite number of potential specializations might be investigated by a partial evaluator. The `polymorphic-double` function doubles its input based on the type of its argument. While `polymorphic-double` might be considered a trivial example, it is representative of code that arises in many programs .

Consider the input program in Figure 2.2 that might call `polymorphic-double` with the following different values for its argument: an unknown list, the string `"ab"`, an unknown string, and the integers `1`, `2`, `3`, `4`, ..., up to some unknown limit. A partial evaluator could create many different potential specializations. In striving to maximize runtime performance, it might produce potential specializations for the unknown list, the string `"ab"`, the unknown string, and some finite set of integers. However, if termination is to be insured, the partial evaluator must not attempt to create potential specializations for the infinite set of positive integers. At some point it must stop and generate a potential specialization for the unknown integer.

Alternatively, the partial evaluator could chose to be more conservative and generate fewer potential specializations. It could generate the potential specialization for the unknown string and use that potential specialization for the string `"ab"`, as well. It could generate one potential specialization for the unknown integer instead of several potential specializations for different integers. In the extreme, it might generate only a single potential specialization for the unknown argument and use it throughout.

```
(define polymorphic-double-consumer
  (lambda (lst str int)
    (polymorphic-double lst)
    . . .
    (polymorphic-double "ab")
    . . .
    (polymorphic-double str)
    . . .
    (let loop ((i 1))
       (if (< i int)
           ( ... (polymorphic-double i) ...
            (loop (1+ i))))))
```

Figure 2.2: A program calls `polymorphic-double` with a number of different types of arguments

As has been previously explained, there is a tension between the desire to insure termination by maintaining finiteness and the need to investigate enough of the appropriate potential specializations in order to generate high performance residual code. There are a plethora of termination mechanisms that could be used to insuring finiteness. The simplest of these is just to place a finite limit on the number of potential specializations of each function to investigate. However, the key is to find a termination mechanism that achieves finiteness while artfully selecting the potential specializations to investigate. This turns out to be significantly more difficult.

2.2 Equivalence Classes

It is instructive to think about the problem of selecting a finite number of potential specializations of each function in terms of equivalence classes. This approach is advocated by Neil Jones in [29]. If all possible applications of each function are divided into a finite set of disjoint equivalence classes, at most a finite number of potential specialization are created for each equivalence class, and a program consists of a finite number of functions, then a partial evaluator is guaranteed to create a finite number of potential specializations.

I define equivalence of applications constructively based on equivalence of their constituent components. Two applications are equivalent if the applied functions are equivalent and the argument sets to which the functions are applied are equivalent. For the purposes of this discussion, the nuances of function equivalence will not be considered. For the time being, a very simple model in which only identical functions are equivalent and all others are distinct suffices. This presentation concentrates on argument equivalence.

For simplicity, I will begin by considering equivalence classes for applications of functions of a single argument. For each function of one argument, there are an infinite number of argument values to which the function could be applied. Imagine partitioning these values into a finite number of sets. If a partial evaluator creates at most one potential specialization of a function for each of the finite number of argument sets, the partial evaluator is guaranteed to investigate only a finite number of potential specializations of each function.

Any finite *partitioning* of argument values into disjoint sets serves to guarantee termination; however, some partitionings yield far better residual code than others. A partial evaluator limited to investigating at most one potential specialization for each partition must produce potential specializations correct for all values in each partition. In other words, any optimizations performed in creating the potential specializations must be correct for all values in the corresponding partition. This motivates the desire to create partitions in which all the values share properties that enable the same optimizations to be performed by a partial evaluator. If an optimization is correct for one half of the values in a partition, but not for the other half, then improved potential specializations would result from subdividing the existing partition into two separate partitions.

Effective partitionings of argument values to a function might aptly be regarded as separation of a language's domain of values into equivalence classes based on the function being applied and the optimizations performed by a partial evaluator. Ideally, the values in each partition ought to be "equivalent" in the respect that the partial evaluator is able to perform the identical optimizations on the body of the function for every one of the values. This allows the most highly optimized potential specializations to be created and used for each possible argument value.

Consider once again the `polymorphic-double` function in Figure 2.1 on page 8.

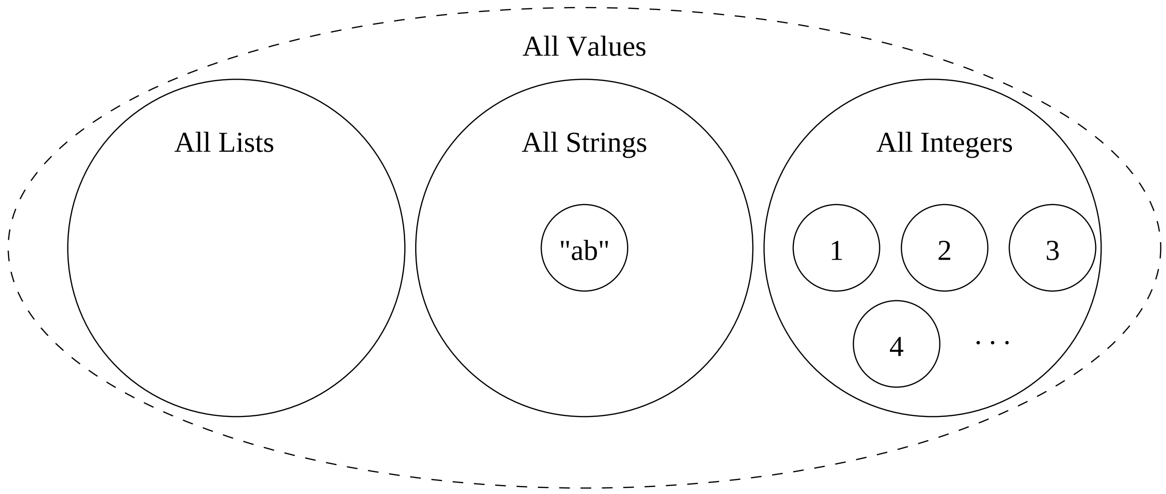

Figure 2.3: `polymorphic-double` arguments

The sets of values to which `polymorphic-double` was to be applied were an unknown list, the string `"ab"`, an unknown string, and the integers `1`, `2`, `3`, `4`, ..., up to some unknown limit.[1] These values can be represented by the Venn diagram shown in Figure 2.3. Note, the set of all values has been shown in the diagram with a dotted line since there was no application of `polymorphic-double` to a completely unknown value. The set of all values has been included as a reference to show the partitioning of the complete domain of values of the language.

The finite set of equivalence classes for the argument of `polymorphic-double` shown in Figure 2.4 guarantees termination of partial evaluation of a program utilizing `polymorphic-double` as specified in this example. The partial evaluator would create a potential specialization for each of the six equivalence classes with solid boundaries.

Figures 2.5 and 2.6 show successively coarser grained partitionings of the domain of all possible argument values. In Figure 2.5, there is one single equivalence class for all of the integers. A partial evaluator using the equivalence classes in Figure 2.5 would not create any potential specializations for any designated integers. In Figure 2.6, the separate equivalence classes for unknown lists and unknown strings have been eliminated. Since these sets have been subsumed into the set of all values, a potential

[1]It has been assumed that we know about all applications of `polymorphic-double`. This is often referred to as the *closed-world* assumption.

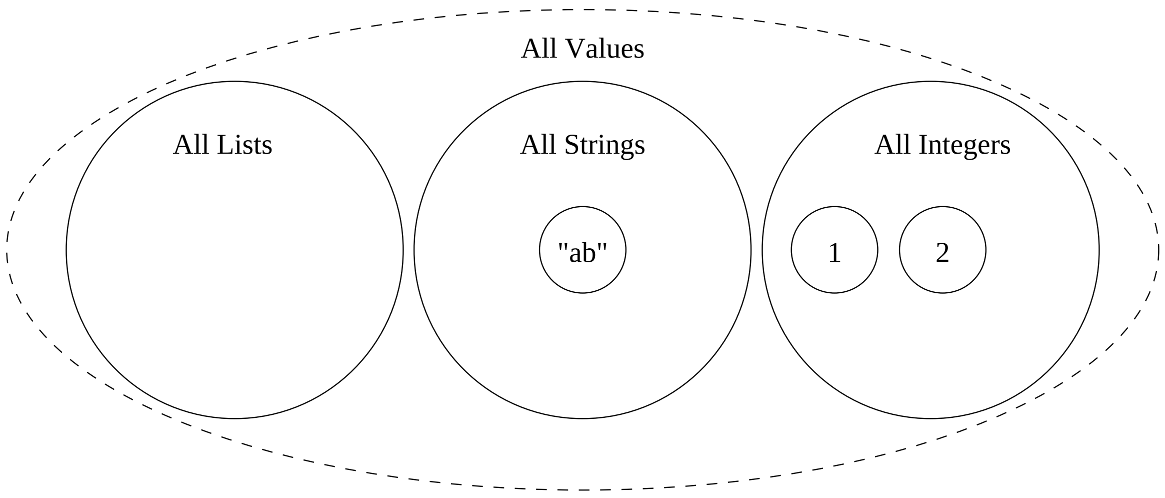

Figure 2.4: Equivalence classes for `polymorphic-double`

specialization is necessary in this case for the completely unknown argument.

The case of guaranteeing a finite number of potential specializations for a function of multiple arguments is quite similar to that for a function of a single argument. The multiple arguments can be considered to be an n-tuple, with n being the arity of the function. So long as the set of all possible n-tuples is partitioned into a finite number of disjoint equivalence classes for each function, it is guaranteed that a finite number of potential specializations will result.

The partitioning of n-tuples can be performed in many ways. One simple approach it to form separate equivalence classes for each of the arguments to a function. Equivalence classes for the set of n-tuples can then be defined based on the cross product of the equivalence classes for each of the arguments. So long as there are a finite number of equivalence classes for each of the arguments of a function, there will result a finite number of equivalence classes for n-tuples of arguments to the function.

2.2.1 How Existing Partial Evaluators Select Equivalence Classes

Partial evaluators have been divided into two separate categories in the literature: *offline* and *online*. The use of equivalence classes by the termination mechanisms for these two types of partial evaluators is discussed separately.

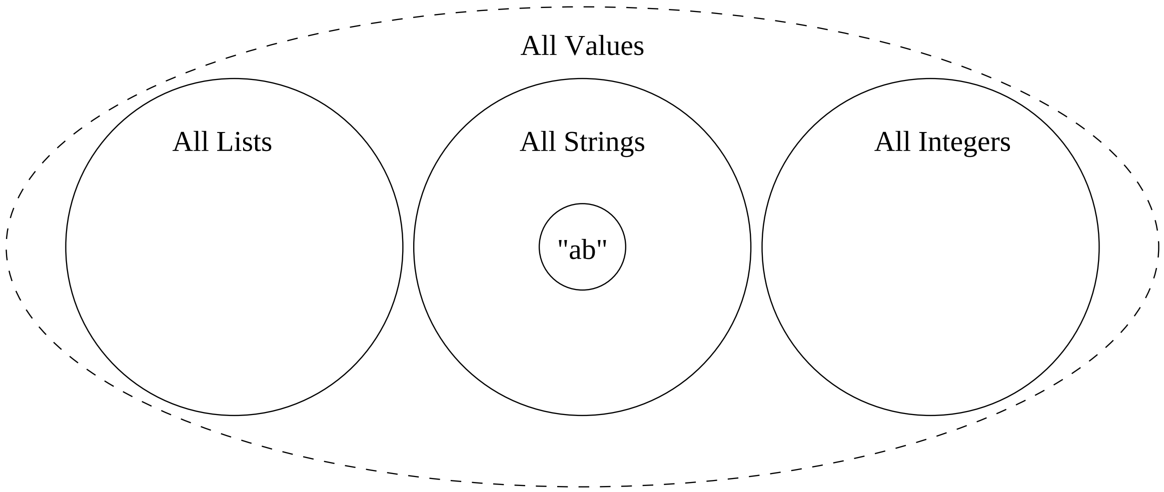

Figure 2.5: Coarser grained equivalence classes for `polymorphic-double`

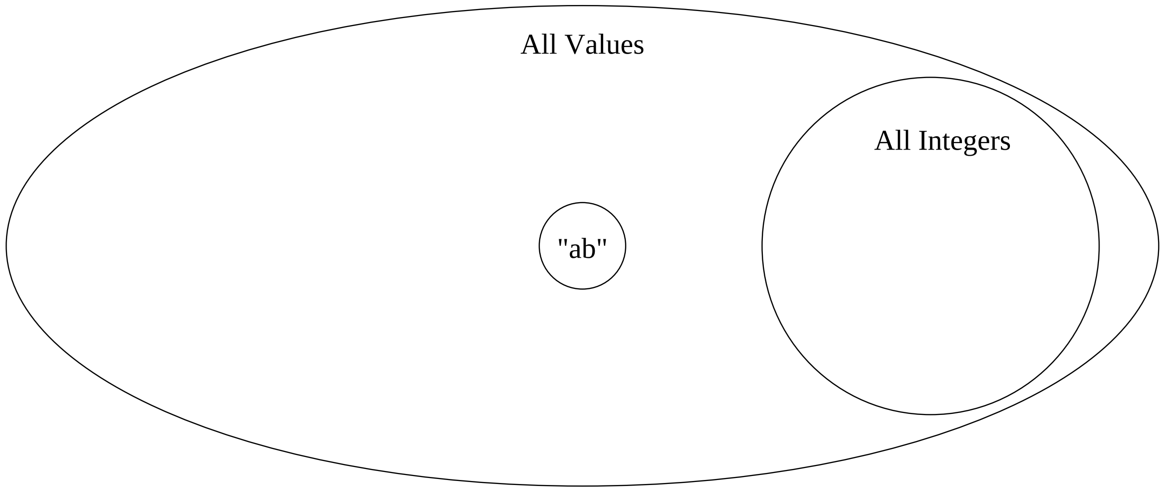

Figure 2.6: Even coarser grained equivalence classes for `polymorphic-double`

Offline

Offline partial evaluators are characterized by a prepass called Binding Time Analysis (BTA)[29]. The purpose of BTA is to create templates for the equivalence classes to be used during partial evaluation. While different BTA's create different templates yielding different equivalence classes, they all share a number of common properties described briefly below.

All BTA's generate templates based on separating arguments to functions into two classes: *static* and *dynamic*. The first and simplest BTA's were later to be called *monovariant* BTA's. A monovariant BTA selects a single labeling of the arguments of each function of each program as static or dynamic. The dynamic arguments are ignored when defining equivalence classes. A different equivalence class can be created for application of a function to each possible combination of values for the argument positions labeled as static.

For example, consider the length program in Figure 2.7. There are four possible templates due to the four possible combinations of labelings of the two arguments to the function `loop` as either static or dynamic. If both `lst` and `ans` are labeled as dynamic, then only a single equivalence class results for applications of `loop`. At most one specialization of `loop` will exist in the residual program. If `lst` is labeled as dynamic and `ans` is labeled as static, then a different equivalence class can exist for every possible value of `ans`. This corresponds to having a potential specialization of `loop` for counting each successive element of lists. One for the first element, one for the second, etc. Since this template can produce an infinite number of equivalence classes, an infinite number of potential specializations might result.

Alternatively, if `lst` is labeled as static and `ans` is labeled as dynamic, then a different equivalence class can exist for every possible value of `lst`. This corresponds to having a potential specialization of `loop` for computing the remaining length of each different list passed to `loop`. If `length` were called on two different length lists sharing a common tail, the potential specializations of `loop` for the two tails would be identical. Equivalence classes for applications of `loop` in this case are only being defined based on the value of `lst`, not based on the length of the portion of the list already analyzed and stored in `ans`. Once again, this template can produce an infinite

```
(define length
  (lambda (lst)
    (loop lst 0)))

(define loop
  (lambda (lst ans)
    (if (null? lst)
        ans
        (loop (cdr lst) (1+ ans)))))
```

Figure 2.7: A function for computing the length of lists

number of equivalence classes so an infinite number of potential specializations might result.

Finally, if both `lst` and `ans` are labeled as static, then a different equivalence class can exist for every different combination of the arguments. This could lead to the greatest variety of potential specializations. However, based on the infinite number of equivalence classes resulting, it might yield divergence of the partial evaluator.

Templates that can yield an infinite number of equivalence classes might or might not produce an infinite number of potential specializations in practice. Consider a program in which the only two applications of `length` are `(length '(a b))` and `(length '(z a b)`. The template where `lst` is static and `ans` is dynamic would likely produce potential specializations for the fours values of `lst`: `(z a b)`, `(a b)`, `(b)`, and `()`. A template with both `lst` and `ans` static would likely produce potential specializations for the seven possible combinations of the values for `lst` and `ans`: {`(z a b)`,0}, {`(a b)`,1}, {`(b)`,2}, {`()`,3}, {`(a b)`,0}, {`(b)`,1}, and {`()`,2}.

Nearly all templates contain at least one static argument. When the static argument has a potentially inifinte set of values it can assume, an infinite number of equivalence classes can result. The key to designing an effective BTA is generating templates for functions that yield a terminating partial evaluation despite the potentially infinite set of equivalence classes that could in theory be instantiated from those templates.

In the simplest BTA model, a static argument is one whose value is always known during partial evaluation for every function call. A dynamic argument is ones whose value might, or might not be, known. If an argument's value is sometimes known and other times unknown then it is labeled as dynamic in order to maintain control over the number of equivalence classes. This ideal model of BTA often yields templates producing an infinite number of potential specializations during partial evaluation. To avoid divergence, most BTA's replace some static labels with dynamics in order to constrain the set of possible equivalence classes (e.g., [40] and [8]).

Finer grain control over the equivalence classes produced by BTA has been achieved in two ways. *Polyvariant* BTA potentially creates multiple different templates for the same function by assigning multiple different labelings to the arguments to a function. Different templates are utilized for different call sites[29]. *Partially static* BTA is based on a richer set of labelings. Arguments used to store composite values like pairs, lists, and vectors can be labeled as partially static and partially dynamic, designating which parts of the composite data are to be used in defining equivalence classes and which parts are to be ignored[21]. For example, these richer templates can represent equivalence classes distinguished based on the values of the cars of pairs, and not their cdrs.

Online

Online partial evaluators differ from offline partial evaluators by defining their equivalence classes on the fly as the partial evaluation progresses. I will discuss only those online systems that automatically generate their equivalence classes. I do not discuss online systems that base the selection of equivalence classes on user supplied annotations [23] and declarations in a metalanguage [11].

When online partial evaluation is initiated, there is effectively a single potential specialization for each source function: the function itself. It is associated with a single equivalence class containing all possible argument values. As online partial evaluation proceeds, additional equivalence classes are created, along with their associated potential specializations. The new equivalence classes are formed by successively subdividing the initial equivalence class for each function. Termination of the partial evaluator is dependent on ensuring only a finite number of divisions of the equivalence class for each function take place. Finiteness is maintained by detecting recursions generating a possibly never-ending series of subdivisions of an equivalence

```
(define car
  (lambda (val)
    (if (pair? val)
        (primitive-car val)
        (error "Illegal argument supplied to car:" val))))
```

Figure 2.8: An implementation of the `car` function

class and disallowing further division.

2.2.2 Improved Notions of Equivalence

This section presents a new basis for defining equivalence classes for partial evaluation. It begins by introducing the concept of *use of information* and goes on to present Ruf's *domain of specialization.* It concludes by describing a refined version of the domain of specialization.

Use of Information

The equivalence classes presented so far have been based on the values of arguments to a function. I propose equivalence classes based on how the values of arguments are used in creating specializations. To be more precise, I propose distinguishing amongst equivalence classes based on the information content in argument values utilized in performing optimizations to create specializations.

For example, consider the implementation of the `car` function shown in Figure 2.8. Traditional partial evaluators would place `(car '(a b))` and `(car '(a c))` in different equivalence classes since the two arguments are distinct. However, the specializations produced in the two cases are identical since the same optimizations can be performed in both cases.

To better understand why, consider the optimizations that can be performed in the two cases. First, the expression `(pair? val)` can be evaluated to yield `#t`. Next, the conditional can be replaced by its consequent, `(primitive-car val)`. Finally, the `primitive-car` operation can be performed to yield `'a`.

Another example of a definition of equivalence classes based on the information

```
(define not
  (lambda (arg)
    (if (boolean? arg)
        (if (eq? arg #t)
            #f
            #t)
        #f)))
```

Figure 2.9: An implementation of the `not` function

utilized in creating specializations is presented by the implementation of the `not` function shown in Figure 2.9. Consider specializations of `not` generated for the expressions `(not 1)` and `(not 3)`. In both cases the result is a function that always returns `#f`; however, traditional partial evaluators would place the two potential specializations into separate equivalence classes since the applications of `not` are to two different values. I propose the two applications ought to be part of the same equivalence class since the information utilized in performing optimizations in both cases, that the argument is not a boolean, is identical.

What precisely do I mean by information used in performing optimizations to generate potential specialization? I mean information necessary to perform delta reductions (execute primitives) or execute conditional control flow operators to replace an expression by a simpler expression or value. This does not include the substitution of values for variables in beta reductions (function applications) performed during partial evaluation. No information is utilized in performing a substitution of a value for a variable, only in performing a computation utilizing a value.

For an example of the distinction between information that is used in performing delta reductions and the case when values are substituted for variables without any information being used, consider the function `polymorphic-+` in Figure 2.10. `polymorphic-+` is a cousin of `polymorphic-double` that performs either addition, string concatenation, or appends two lists depending on the types of its arguments. Partial evaluation of the application of `polymorphic-+` to the value `1` and an unknown value, utilizing equivalence classes based only on information used in performing delta reductions, yields the potential specialization in Figure 2.11, associated with

```
(define polymorphic-+
  (lambda (arg1 arg2)
    (cond
     ((and (number? arg1) (number? arg2))
      (+ arg1 arg2))
     ((and (string? arg1) (string? arg2))
      (string-append arg1 arg2))
     ((and (list? arg1) (list? arg2))
      (append arg1 arg2))
     (else (error "Bad polymorphic-+:" arg1 arg2)))))
```

Figure 2.10: A polymorphic `+` function that operates on numbers, strings, and lists

```
(define polymorphic-+
  (lambda (arg1 arg2)
    (cond
     (number? arg2)
      (+ arg1 arg2))
     (else (error "Bad polymorphic-+:" arg1 arg2)))))
```

Figure 2.11: A specialization of `(polymorphic-+ 1 unknown)` using equivalence classes based on information used in performing delta reductions

the equivalence class in which only the information that `1` is a number is used. If substitution of values for variables were a type of use reflected in the equivalence classes utilized, the potential specialization in Figure 2.12 would result for the equivalence class based on use of the value of `1`. The difference between the two potential specializations is the substitution of the value `1` for the variable `arg1` in the applications of `+` and `error` in the second potential specialization.

The simple reason use based equivalence classes do not reflect values substituted for variables is that the smaller equivalence classes that would result tend to cause a partial evaluator based on those equivalence classes to diverge. Information used in performing delta reductions distinguishes whether applications are equivalent in a more fundamental sense. The more polymorphic potential specializations resulting from the larger equivalence classes reflect differences amongst iterations of a recursion that determine whether partial evaluation of the recursion will complete on its own

```
(define polymorphic-+
  (lambda (arg1 arg2)
    (cond
     (number? arg2)
      (+ 1 arg2))
     (else (error "Bad polymorphic-+:" 1 arg2)))))
```

Figure 2.12: A specialization of (`polymorphic-+ 1 unknown`) based on information used in delta reductions and the substitution of values for variables

or needs to be terminated to prevent divergence.

Delayed commitment partial evaluation postpones decisions whether to substitute actuals for formals in potential specializations until the code generation phase. This means the specialization in Figure 2.12 might appear in the final residual code even though the equivalence classes utilized for termination are associated with the potential specialization in Figure 2.11. The code generation phase decides whether substitutions of values for variables yielding only slightly higher performance specializations come at the expense of requiring a greater number of different specializations, resulting in unacceptable growth in code size.

Ruf's Domain of Specialization

Utilizing my original formulation of equivalence classes based on the information in arguments that is *used to construct* specializations, Ruf showed one could in practice enlarge the equivalence classes beyond the size generated by most partial evaluators [38, 37]. He further demonstrated that larger equivalence classes enable significantly increased reuse and sharing in residual code, without degrading the quality of the code. Ruf coined the term *domain of specialization* (DOS) to characterize the equivalence class of applications in which a specialization can be safely reused, and explained how to compute a safe approximation to the DOS (called the Most General Index) by tracking the information used in constructing a specialization. Ruf did not use the DOS as the basis of his termination strategy.

Yet a Better DOS

Equivalence classes of applications based on use of information in performing optimizations assumes that optimizations performed in creating different potential specializations differentiate them. This approach misses one important fact: only those computations a function performs in order to produce its result are significant. Any other computations are irrelevant. Therefore, use equivalence ought only be based on uses of values that allow relevant computations to be optimized. Furthermore, the only significant computations in a complete program are those contributing to computing the result of the program. Computations that create intermediate results not utilized in later relevant computation are also irrelevant.

Equivalence classes based only on information that *causally contributes to generating the result of a program* can be somewhat larger than those based on the DOS defined by Ruf. I believe these larger equivalence classes to be a more effective basis for making termination decisions. But, what precisely does it mean to base equivalence classes on information used in performing optimizations that contribute to the computation of the result of a program?

The less information used about the return value of a specialization applied in some context, the less constraints that are placed on which potential specializations could correctly be utilized at that application site. If one thinks of all the potential specializations that can correctly be utilized at some application site as members of an equivalence class, then including consideration of the information used about the return value of the specializations in that context effectively increases the size of the equivalence class of specializations that can be called from that application site. For example, assume the function `test` in figure 2.13 is specialized on the number `5`. The DOS for `square` would be that `n` must be `5`, which is reasonable since the entire body of `square` could be replaced by the residual constant `25`. However, the DOS for `test` would specify that `q` must be `5`. In fact, the entire body of `test` could be replaced by the residual constant `#t`. This specialization of `test` is valid whenever `q` is a number. The DOS has defined an overly restrictive equivalence class for the specialization of `test` applied to `5`.

Weise and I in [31] chose to refer to the DOS defined by Ruf as the *context free*

```
(define square
  (lambda (n)
    (* n n)))

(define test
  (lambda (q)
    (number?
     (square
      q))))
```

Figure 2.13: DOS defines too small equivalence classes

domain of specialization (CF-DOS) to differentiate it from a *context sensitive domain of specialization* (CS-DOS) that accounts for both the values to which a specialization is applicable and how its result is used. When a function is called from a context that doesn't use all the information available about its return value, it is important to know if a specialization already exists (even one returning a different value) that can correctly be applied at the new call site. The CF-DOS is not as useful for this purpose since it is a characterization based on information not necessarily used in every context. In other words, equivalence classes based on the CF-DOS may be unnecessarily small; in particular, too small to ensure effective termination.

For example, consider the "counting up" factorial program in Figure 2.14. Assuming the value of `n` is unknown, during partial evaluation a series of iterations of `loop` are analyzed starting with `i` equal to `1` and proceeding through successive integers. The CF-DOS of `loop` for each iteration is based on the values of `i` and `ans` for the iteration since the computation `(1+ i)` utilizes the value of `i` and the computation `(* ans i)` utilizes the values of both `i` and `ans`. Since equivalence classes based on the CF-DOS differ for every iteration of `loop`, an infinite number of iterations are investigated and partial evaluation diverges. However, the CS-DOS for the iterations of `loop` are all the same. Neither the values of `i` nor `ans` are utilized in computing the return value of the factorial program so long as the value of `n` is unknown. Since all iterations belong to the same equivalence class, partial evaluation terminates when utilizing equivalence classes based on the CS-DOS.

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (let loop ((i 1)
                   (ans 1))
          (if (> i n)
              ans
              (loop
               (1+ i)
               (* ans i)))))))
```

Figure 2.14: “Counting up” factorial program

The computational complexity of producing potential specializations for equivalence classes based on approximations to the CF-DOS is much less than using the CS-DOS. Whereas the CF-DOS requires investigating different potential specializations for different argument sets, the CS-DOS requires investigating different potential specializations even for identical argument sets when the applications appear in different contexts, which is an exponentially larger set of potential specializations. Furthermore, acquiring the information about contexts needed for equivalence classes based on approximations to the CS-DOS often requires analyzing the same context (i.e., continuation) separately for each flow of control into the context (i.e., invoking the continuation).

As will be discussed in Sections 2.4 and 2.6, eager use analysis computes an approximation to the CF-DOS, and lazy use analysis computes an approximation to the CS-DOS. The role of CPS conversion in lazy use analysis and other partial evaluators in order to improve context sensitivity is presented in Section 4.2. The description of use analysis in this chapter for simplicity discusses use analysis without CPS conversion. The low level description of the analysis including CPS conversion appears in Chapter 3. Finally, the detrimental performance implications for lazy use analysis resulting from the need to maintain context sensitivity are discussed in Chapter 5.

2.2.3 Higher-Order, First-Class Functions and Equivalence

Higher-order, first-class functions add new complications to the definition of equivalence classes. Should two closures formed from the same lambda expression during symbolic execution ever be members of the same equivalence class? Should applications of different closures of the same lambda expression ever be deemed equivalent?

I believe the answer to the first question must be yes in order to yield effective termination and produce high quality residual code. The answer to the second question depends on the type of termination mechanism being utilized. This section begins with an example in which prohibiting equivalence between different closures of the same lambda expression presents a termination problem. It then presents an example demonstrating that always assuming two closures formed from the same lambda expression are equivalent can yield code quality and correctness problems. The final example shows that equivalence of applications of different closures of the same lambda expression can be desirable. The section concludes with a discussion of how I propose closure equivalence ought to be handled when equivalence classes are based on use of information.

When Lack of Equivalence Presents Problems

Utilization of higher-order, first-class functions as arguments can present a termination problem unless different closures formed from the same lambda expression can be equivalent. Consider the canonical recursive function shown in Figure 2.15. It can be transformed into a version using closures as arguments as shown in Figure 2.16 by converting each argument at each call site into a thunk returning the original argument value, by dethunking the arguments at each reference site by applying the transformed arguments to no value, and by creating the obvious helper function that maintains the original interface.

Applying this same transformation to the factorial function in Figure 2.17 yields the version in Figure 2.18. Consider partial evaluation of the transformed version of `fact` in Figure 2.18 applied to an unknown integer. During partial evaluation, the argument to every application of `closure-arg-fact` is a different closure returning an unknown integer. If none of these closures are ever considered equivalent to each

```
(define func
  (lambda (arg)
    . . .
    (func value)
    . . .
        arg
    . . .
    ))
```

Figure 2.15: A canonical recursive function

```
(define func
  (lambda (arg)
    (closure-arg-func (lambda () arg))))

(define closure-arg-func
  (lambda (arg)
    . . .
    (closure-arg-func (lambda () value))
    . . .
        (arg)
    . . .
    ))
```

Figure 2.16: A canonical recursive function using closures for arguments

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (-1+ n))))))
```

Figure 2.17: Factorial program

```
(define fact
  (lambda (n)
    (closure-arg-fact (lambda () n))))

(define closure-arg-fact
  (lambda (n)
    (if (zero? (n))
        1
        (* (n) (closure-arg-fact (lambda () (-1+ (n))))))))
```

Figure 2.18: Factorial program using closures for arguments

other, partial evaluation fails to terminate.

When Equivalence Presents Problems

Just as it is sometimes critical that different closures formed from the same lambda expression be considered equivalent, it is also desirable in some cases that they not be deemed equivalent. Consider another partial evaluation of the transformed version of factorial in Figure 2.18, this time applied to the value `5`. If all closures formed from the same lambda expression are always deemed equivalent then every iteration of `closure-arg-fact` after the first is deemed equivalent. This causes termination of the recursion before the factorial of 5 is computed. The result is a residual recursion, as opposed to a residual program just returning 120. This is suboptimal residual code.

Furthermore, considering different closures of the same lambda expression always to be equivalent can lead to the creation of incorrect residual code. Different closures of the same lambda often contain different values in their environments. Using one closure in place of another can result in the utilization of an incorrect value in a

```
(define func
  (lambda (arg)
   ((upward-funarg-func arg))))

(define upward-funarg-func
  (lambda (arg)
    (lambda ()
      . . .
      ((upward-funarg-func value))
      . . .
         arg
      . . .
      )))
```

Figure 2.19: A canonical recursive function utilizing an upward funarg

computation.

Having demonstrated the desirability of allowing two closures of the same lambda expression to belong to the same equivalence class, the next issue is when it is desirable for applications of two closures of the same lambda expression to be deemed equivalent. Consider once again the canonical recursive function shown in Figure 2.15. It can be transformed into a version based on upward funargs as shown in Figure 2.19 through the following process: replace the body of the function with a lambda expression of no arguments whose body is the body of the original function, augment all applications of the function by wrapping them in an application of the returned value to no arguments, and create an new function with the original interface to call the transformed version of the function.

Applying this same transformation to the factorial function in Figure 2.17 yields the new version in Figure 2.20. This program presents an interesting question. Is the function `fact` in Figure 2.20 recursive? If one takes as the definition of a recursive, a function called while another application of the same function is still pending (i.e., has yet to return a value), then `fact` is not recursive. Each time `fact` is called, the previous application of `fact` has already returned a closure as an upward funarg; and, it is the upward funarg that is still executing. Of course, a different closure is returned each time so the upward funargs are only recursive if all the upward funargs

```
(define fact
  (lambda (n)
   ((upward-funarg-fact n))))

(define upward-funarg-fact
  (lambda (n)
    (lambda ()
      (if (zero? n)
          1
          (* n ((upward-funarg-fact (-1+ n))))))))
```

Figure 2.20: Factorial program utilizing an upward funarg

formed from the same lambda expression are considered equivalent for the purpose of defining recursion.

The significance of whether Figure 2.20 contains a recursion depends on the structure of a partial evaluator's termination mechanism. Some termination mechanisms make no distinction between recursive applications and all other applications. The example in Figure 2.20 presents no problem for partial evaluators utilizing these forms of termination mechanisms. However, this example presents a potential problem for termination mechanisms based on detecting equivalent, recursive applications. Use analysis falls in the latter category so the distinction is salient.

Termination mechanisms based on detecting equivalent, recursive applications must allow two applications of different closures of the same lambda expression to be deemed equivalent in order to terminate on the example in Figure 2.20. This enables a termination decision to be made for the 'recursion' of the upward funargs when both the functions applied and their arguments are deemed to be equivalent.

Suggested Means of Handling Equivalence of Functions

I believe closures ought to be handled analogously to aggregates. Lambda is really just a constructor binding together a piece of code and an environment into a single object. Two closures ought to be deemed equivalent if and only if both the code and the environment of the closures are equivalent. Code equivalence is easily defined in terms of closures being formed from the same lambda expression.[2] Again analogous to aggregates, environment equivalence can be defined in terms of equivalence of corresponding bindings in the environments.

If one's equivalence classes are based on use of information, then the following types of uses of closures result. Any time a closure is applied, its code (i.e., the lambda expression from which it is formed) is used in order to execute the body of the function. This means all closures are equivalent so long as they are not applied. As soon as they are applied, only closures of the same lambda are deemed equivalent. Uses of information about values in the environment of a closure result naturally as the body of a closure is executed.

Having defined closure equivalence, application equivalence follows directly. Two applications are equivalent if the applied closures and all of the arguments are equivalent. Another way of thinking about application equivalence is that all the values in the environment of a closure effectively become extra arguments of applications. It should be noted, this conceptualization of application equivalence is analogous to the result of lambda lifting [27] programs prior to partial evaluation.

2.3 Use Analysis: Domains and Lattices[3]

Use analyses calculate an approximation to the information used by a partial evaluator in performing optimizations of expressions. The information *used* in performing optimizations can be represented by elements of domains of types of use and organized into a lattice. This section describes the properties of the domains and lattice utilized by all use analyses. In subsequent sections the presentation of use analysis proceeds in two stages. First, eager use analysis, which is fairly simple to explain and understand, is developed.[4] This is followed by a discussion of the shortcomings of the eager form of the analysis. Finally, an improved lazy use analysis is discussed.

The selection of the domains representing types of use and a lattice based on those domains is a plug replaceable module in the use analysis algorithms to be presented.

[2]Any more complex form of code equivalence rapidly becomes undecidable.
[3]Some portions of this section are taken in whole or part from [31].
[4]The first implementation of eager use analysis is described by Ruf in [37].

The concepts behind use analysis are independent of the precise choice of the domains and lattice. However, the domains and lattice selected can have a pronounced effect on the quality of residual code produced by a partial evaluator since they determine the equivalence classes used in choosing those potential specializations to investigate. In addition, the domains and lattice must exhibit certain monotonicity properties with respect to the primitive operations of the source language, as implemented by delta reductions, in order for use analysis to be meaningful.

Figure 2.21 shows the value domains for a pure subset of Scheme [10] that does not include all the data types in the full Scheme language. The pure subset selected includes the following types of expressions: variable references, literals, function applications, function definitions (`lambda`), conditionals, recursive function definitions (`letrec`), and definitions (`define`). Being a pure subset, it does not include assignments (`set!`), sequencing (`begin`), or the ability to create or throw to first-class continuations.

A richer set of value domains capturing the partial information useful during partial evaluation appears in Figure 2.22. The partial evaluation value domains in Figure 2.22 also happen to serve as one reasonable set of values for representing the types of information that can be used about a value. For example, use of only the integer property of the number `3` during partial evaluation might be represented by the abstract value $\perp_{Int}$.

As with any set of domains selected, there are many types of information about values that cannot be precisely represented by the domains in Figures 2.22. For example, the closest representations of the information that an integer is less than 5 are either the integer's identity or $\perp_{Int}$. The integer's identity is an overspecification of the information used, excluding other integers less than 5. $\perp_{Int}$ is an underspecification, including integers greater than or equal to 5. Whenever no element of a domain can precisely represent the information content used about a value, a choice must be made between either overspecification or underspecification. The choice of how to approximate use can have a significant impact on the size of the equivalence classes utilised by a partial evaluator and therefore on termination and the number

Int	$=$	$\texttt{0} + \pm\texttt{1} + \pm\texttt{2} + \cdots$	integers
$Bool$	$=$	$\texttt{true} + \texttt{false}$	booleans
Sym	$=$	$\texttt{'a} + \texttt{'b} + \cdots$	symbols
Nil	$=$	$\texttt{nil}$	empty list
$Pair$	$=$	$Sval \times Sval$	pairs
$Closure$	$=$	$Lambda \times Env$	closure values
Env	$=$	$(Id \rightarrow Sval)^{\star}$	environments
$Sval$	$=$	$Int + Bool + Nil + Pair + Closure$	scheme values

Figure 2.21: Value domains for a pure subset of Scheme

Int	$=$	$\texttt{0} + \pm\texttt{1} + \pm\texttt{2} + \cdots$	integers
		$\perp_{Int}$	unspecified integer
$Bool$	$=$	$\texttt{true} + \texttt{false}$	booleans
		$\perp_{Bool}$	unspecified boolean
Sym	$=$	$\texttt{'a} + \texttt{'b} + \cdots$	symbols
		$\perp_{Sym}$	unspecified symbol
Nil	$=$	$\texttt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\perp_{Pair} \equiv \perp_{PEval} \times \perp_{PEval}$	unspecified pair
$Closure$	$=$	$Lambda \times Env$	closure values
		$\perp_{Clos}$	unspecified closure
Env	$=$	$(Id \rightarrow PEval)^{\star}$	environments
$Kval$	$=$	$Int + Bool + Nil + Pair + Closure$	known values
$Bots$	$=$	$\perp_{Int} + \perp_{Bool} + \perp_{Pair} + \perp_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \perp_{PEval}$	partial evaluation values
		$\perp_{PEval}$	unspecified value

Figure 2.22: Value domains for partial evaluation of a pure subset of Scheme

of potential specializations investigated.[5]

A transitive binary relation based on the concept of information used about values and denoted by $\prec$ can be defined. $\prec$ should be read as *has less information than* and $\preceq$ as *has less than or the same amount of information as.* A lattice based on the domains in Figure 2.22, the transitive, reflexive binary relation $\preceq$, and a newly

[5]Since an overspecifying use analysis can lead to increased divergence and an underspecifying use analysis can lead to incorrect reuse of potential specializations, it might be desirable to compute two use analyses, one underspecifying use for termination purposes and the other overspecifying use for making code sharing/reuse decisions. I currently only implement the former analysis.

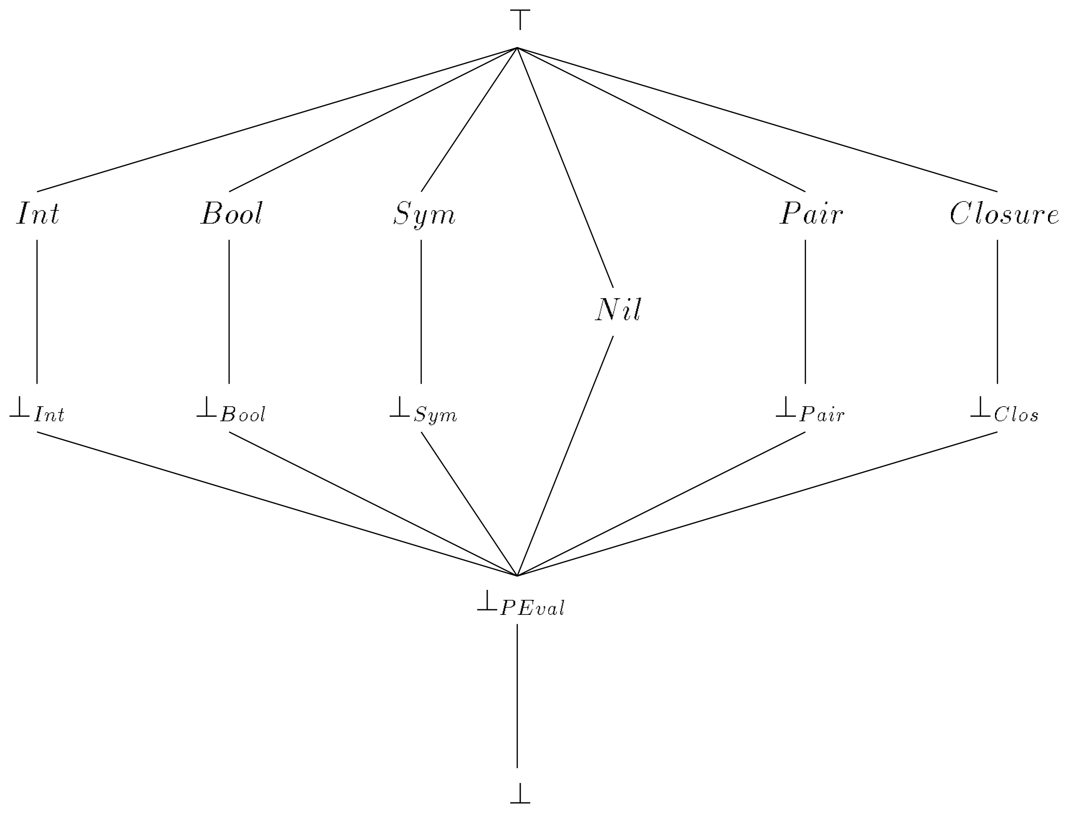

Figure 2.23: Information lattice formed using partial evaluation value domains

introduced top and bottom element appears in Figures 2.23 and 2.24.[6]

The lattice diagram in Figure 2.23 shows that integers, booleans, symbols, the empty list, pairs, and closures are disjoint and incomparable; there is less information in knowing an object's type than in knowing its value (e.g., the relative position of

[6]The orientation of a lattice is arbitrary. Greater information can be represented by points either higher or lower in the lattice. The orientation I selected as presented in Figures 2.23 and 2.24 is opposite of the one used by Ruf in [38].

$$\forall <x,y>, <x',y'> \in Pair.((<x',y'> \preceq <x,y>) \leftrightarrow ((x' \preceq x) \wedge (y' \preceq y)))$$
$$\forall <l,e>, <l',e'> \in Closure.((<l',e'> \preceq <l,e>) \leftrightarrow ((l' = l) \wedge (e' \preceq e)))$$
$$\forall e, e' \in Env.((e' \preceq e) \leftrightarrow (\forall <i,s'> \in e'.(\exists <i,s> \in e.(s' \preceq s))))$$

Figure 2.24: Information lattice equations to supplement the lattice diagram

$\perp_{Int}$ and Int); and there is even less information is knowing one has a value, but not even knowing its type (e.g., the relative position of $\perp_{PEval}$ and $\perp_{Int}$). The top element has no obvious interpretation; however, there is a distinction between $\perp_{PEval}$, representing the object with an unknown value, and the bottom element of the lattice. This distinction will be discussed later in Section 2.6.2 when presenting an algorithm for implementing lazy use analysis.

The equations in Figure 2.24 define the orderings amongst pairs and closures. The pair equation is the typical generalization to aggregates. One aggregate contains less information than another aggregate if the property holds for corresponding components of the aggregates. Closures are organized based on the information used about the variables over which the same lambda expressions are closed. Less or the same amount of information is used about one closure of a lambda expression than about a second closure of the same lambda expression if and only if this property holds for every variable over which the first closure is closed.

2.4 Eager Use Analysis

Eager use analysis is explained in several parts. The data structures used by the analysis are presented. Then, an explanation of how use information is recorded about applications of primitive functions is given. This is followed by a discussion of the core of the analysis with the exception being that the parts of the algorithm relating to termination are omitted. Finally, termination is added to the base algorithm and issues involving and resulting from the termination mechanism are discussed.

Eager use analysis is implemented by an augmented Scheme interpreter. In recognition of the additions to the basic execution model, the augmented interpreter is said to *symbolically execute* the source program. The augmented interpreter operates on the extended value domains of Figure 2.22, rather than the domains for pure Scheme in Figure 2.21. When a control flow decision point is reached for which there is insufficient information during the use analysis phase of partial evaluation to decide what control path will be taken at runtime, the augmented interpreter investigates both possible flows of control. Sometimes the symbolic executor desists investigation of a flow of control because a termination decision is made in order to protect against divergence of the partial evaluator. Finally, symbolic execution collects information about the uses of values and different potential specializations.

The augmented interpreter utilizes several data structures in implementing eager use analysis. These include extended *symbolic values*, *use dependences*, a *stack* of pending function calls, a *table* of potential specializations, a *work list* of flows of control remaining to be analyzed, and *use change daemons* that detect when a new type of use is made of a value. While each of the data structures is explained in more detail later, a brief one sentence description of each is given here as well.

Symbolic values are utilized to represent the information available during partial evaluation about each value on which the augmented interpreter performs computations. This information might include its value (e.g., it is an integer) and other invarients gathered during analysis. Use dependences codify how utilization of information about one data value implies use of some type of information about another data value. For example, when the function application `(boolean #f)` is performed, a use dependence might be created between the symbolic value for the result, `#t`, and the symbolic value for the argument, `#f`, signifying that use of the result value implies use of the property that `#f` is a boolean.

The stack of pending function calls is utilized in determining when a function application is a recursive call. The table of potential specializations is used to determine when a potential specialization already exists for a specific equivalence class. The use analysis algorithm works by placing pieces of program to be analyzed on a work list and removing a new program segment from the work list for analysis each time the algorithm completes work on the current program segment. The algorithm completes when no program segments to be analyzed remain on the work list. Finally, use change daemons detect when a new form of use is made of a value and this change in use might necessitate performing some additional analysis of some program segment. The use change daemon when activated adds an object to the work list to perform the necessary analysis.

Weise utilized symbolic values in Fuse to represent the data values upon which his partial evaluator operated [48]. Weise's symbolic values are composed of two components: a representation of the information available about a value during partial evaluation and a representation of the residual code still needing to be executed at

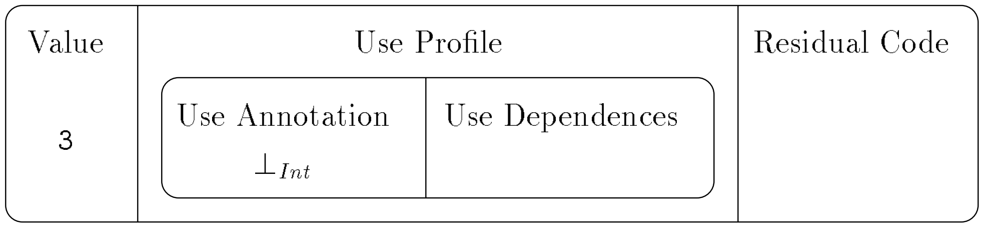

Figure 2.25: A symbolic value representing the value `3` with a use annotation of $\perp_{Int}$

runtime to produce a runtime value from the partial value available during partial evaluation. For use analysis, I have extended symbolic values to include a third component, a representation of the information used about a value in performing delta reductions during symbolic execution. The values stored in the use component will be referred to as *use profiles*. Each use profile is composed of two portions: a *use annotation* recording the information about a value that has been utilized and a list of *use dependences* recording how use of this value implies use of other values. For example, Figure 2.25 shows a symbolic value representing the value `3` with a use annotation of $\perp_{Int}$ and no use dependences or residual code.

Recording the information about argument values used as each delta reduction is performed during symbolic execution is achieved by updating the use annotations of the arguments. Use analysis needs to know how much information about every value is utilized in symbolically executing some expression or program segment, so it records the maximum amount of information utilized about each value during symbolic execution. Every symbolic value is created with an initial use annotation of $\perp$, representing an as yet unused value. Each delta reduction makes some type of use of each of its arguments. The use annotations of the symbolic values for the arguments are updated by computing the *least upper bound* (LUB) of the previous use annotation and the new use made by the current delta reduction.[7]

A table containing a number of applications of primitive Scheme function and the use annotations generated for the underlined arguments appears in Figure 2.26. The use annotations presented are based on the lattice in Figure 2.22. For each of the examples shown, a use annotation precisely capturing the information utilized in

[7]The LUB's are with respect to the lattice of uses being utilized by the use analysis.

Expression	Argument Use Profiles
(integer? $\underline{3}$)	$\perp_{Int}$
(car '$\underline{(1\ .\ 2)}$)	$\perp_{Pair}$
(boolean? $\underline{\perp_{Bool}}$)	$\perp_{Bool}$
(+ $\underline{1}$ $\underline{2}$)	1,2
(* $\underline{3}$ $\underline{0}$)	$\perp_{\texttt{Int}}$,0

Figure 2.26: Eager use annotations

performing the delta reductions is available.

The final two examples in Figure 2.26 demonstrate some of the subtleties of use analysis. Calculating use is in one regard very similar to computing a partial derivative. The information about a single argument used in performing a computation is determined by holding all other arguments constant and observing the dependence of the result on the single input. In the case of `(+ 1 2)`, if the second argument is held constant, the information that the first argument is `1` is necessary in order to compute the result of `3`. Holding the first argument constant, a similar result is obtained for the second argument.

If the computation had been `(< 1 5)`, no use from the domains in Figure 2.22 could precisely capture the information utilized in performing the computation. Assuming the second argument is held constant, there are still many different values for the first argument producing the same result of `#t`. This application is an example of the case discussed in the previous section in which either an underspecification or an overspecification of use is required.

The last example in Figure 2.26 demonstrates that some primitives ought to operate in a special fashion for some argument values. Since anything multiplied by zero produces a result of zero, the value of the first argument in `(* 3 0)` is not needed in order to compute the result. However, since multiplication in Scheme does a type check of its arguments before performing the actual computation, the fact that `3` is an integer is used in executing the `*` function. If Scheme were statically typed, a use of $\perp$ would be generated for `3`.

The `double-integer` function in Figure 2.27 demonstrates how use annotations are calculated utilizing least upper bounds. When `double-integer` is applied to the value `2`, the symbolic value for `val` has an initial value of `2` and use annotation of

```
(define double-integer
  (lambda (val)
    (if (integer? val)
        (* val 2)
        (error "Illegal argument supplied to double-integer:" val))))
```

Figure 2.27: A function for doubling integers

$\perp$. The `integer?` delta reduction creates a use annotation of $\perp_{Int}$ for its argument. Taking the LUB of $\perp$ and $\perp_{Int}$ produces a new use annotation of $\perp_{Int}$ for `val`. Next, the `*` delta reduction produces a use annotation of `2` for `val`. Taking the LUB of the previous use annotation for `val`, $\perp_{Int}$, and the new use, `2`, produces a new use annotation of `2`. The net result is the `double-integer` function produces a use annotation of `2` for its argument when applied to the value `2`.

Nested function applications and aggregates as arguments can produce more complex use annotations. For example, `(integer? (cdr (car '((1 . 2) (3 . 4)))))` results in a use annotation of $<< \perp, \perp_{Int} >, \perp >$ for the argument to `car`. The innermost `car` uses the information the argument is a pair. The `cdr` uses the information the first component of the pair is also a pair. `integer?` uses the information the second component of that pair is an integer. The aggregate argument utilized in this example demonstrates one subtlety of use analysis. The `car` delta reduction extracts one component of an aggregate. The `cdr` delta reduction makes use of information about the extracted component. However, it is also by implication making use of information about the original aggregate. The analysis must insure the appropriate use annotations are created for both the original aggregate and for the substructure returned by the application of `car`.

Figures 2.28-2.31 show how symbolic values, use profiles, and use dependences are utilized to compute the uses described in the previous paragraph. When `car` is applied to a symbolic value representing an aggregate, a new symbolic value and use profile are created for the extracted portion of the aggregate. In addition, a *use dependence* (shown as a dashed line) is created between the use profiles of the new symbolic value and the one representing the aggregate. The use dependence

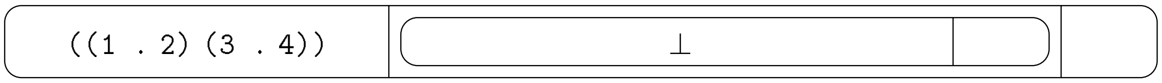

Figure 2.28: Symbolic value for `'((1 . 2) (3 . 4))`

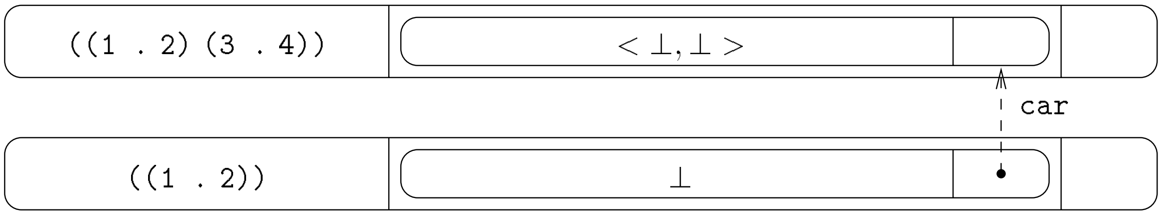

Figure 2.29: Symbolic values for `(car '((1 . 2) (3 . 4)))`

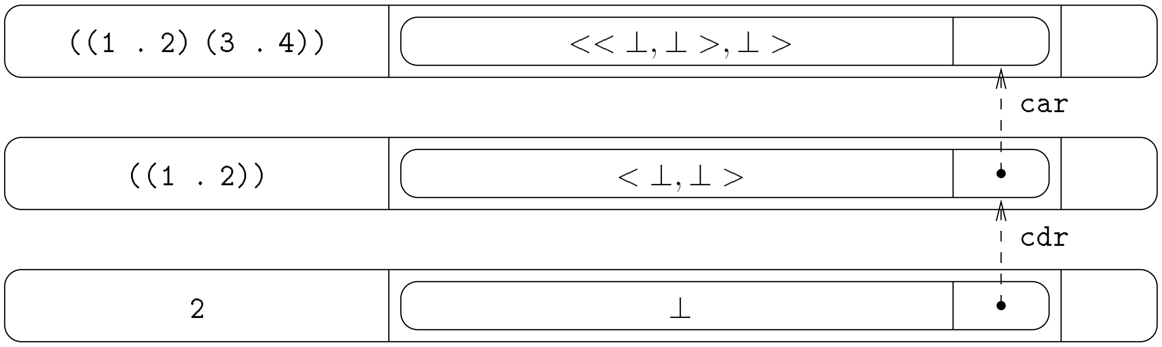

Figure 2.30: Symbolic values for `(cdr (car '((1 . 2) (3 . 4))))`

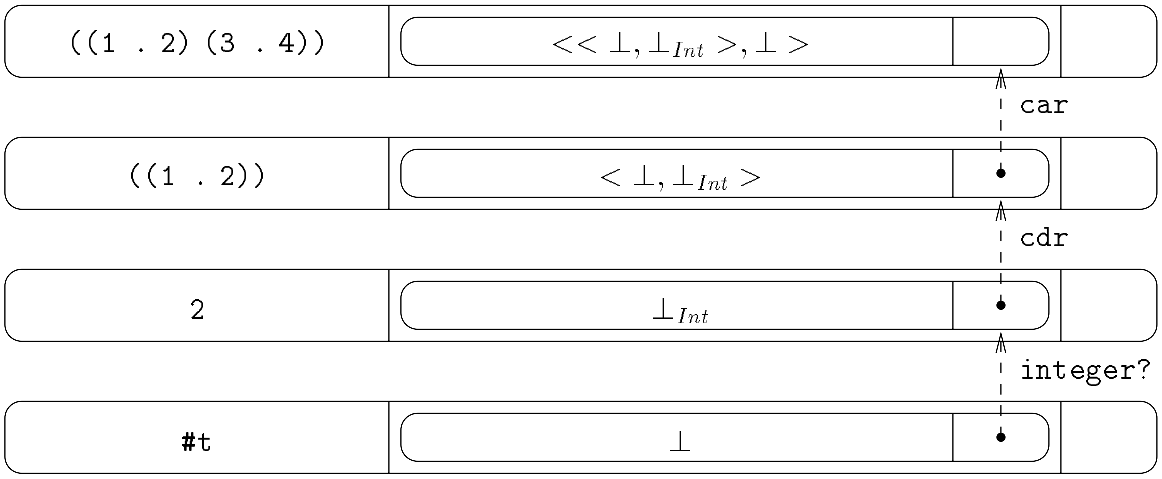

Figure 2.31: Symbolic values for `(integer? (cdr (car '((1 . 2) (3 . 4)))))`

represents that any change in the use annotation of the extracted component must also be reflected by a corresponding change in the use annotation for the aggregate. The use dependences are labeled to indicate how the use of information about one symbolic value implies use of information about another symbolic value. Note, the `car` and `cdr` use dependences affect different components of use profiles at the arrow ends of the dependences.

2.4.1 The Algorithm

Partial evaluation and eager use analysis both begin with a program to be analyzed/specialized and an input specification for that program. The input to the analysis can be represented as a function application of the function initiating execution of the program to be analyzed applied to the input specification, as represented by the arguments. Eager use analysis begins with symbolic execution of this initial application.

In explaining the eager use analysis algorithm, I will separate the issues of how use is computed and potential specializations are analyzed through symbolic execution from the issue of termination. As such, eager use analysis is initially explained assuming the algorithm completes somehow. Termination is added to the algorithm in Section 2.4.2.

I first present how each of the expression types is handled during symbolic execution. The different types of expressions present in the pure subset of Scheme to which use analysis has been applied are variable references, literals, function applications, function definitions (`lambda`), conditionals, recursive function definitions (`letrec`), and definitions (`define`).

Variable references look up a value in an environment. The symbolic execution engine operates similar to a normal evaluator except that a symbolic value is returned by a variable reference instead of a simple value.

Literals are self evaluating and produce a symbolic value whose value is that of the literal. The symbolic value for each literal is initialized with a use annotation of $\perp$ and no use dependences.

Lambda expressions are also self evaluating. They yield a symbolic value whose value is a closure composed of the associated lambda expression and the current lexical environment. As for other self evaluating objects, the symbolic value is initialized with a use annotation of $\perp$ and no use dependences.

Top level definitions bind a name in the global environment to the symbolic value passed as the second argument to `define`. The current implementation does not support Scheme's optional internal definitions.

A discussion of the `letrec` special form is postponed until Chapter 3. As is explained in more detail there, recursive function definitions are syntactically rewritten into code using only the other expression types by utilizing the applicative order Y combinator [34, 20].

This brings us to the two most interesting expression types: function applications and conditionals. Function application depends on whether the function is a primitive or non-primitive function. The simpler case of delta reductions performed by primitive functions is addressed first.

The implementation of each primitive function is built into the symbolic execution engine. Symbolic execution of the application of a primitive function consist of three parts: computation of the symbolic result, assertion of use of the information about the arguments utilized in computing the symbolic result, and creation of use dependences between the symbolic value for the result and the symbolic values for the arguments. Since the details of the implementation of primitive applications was already discussed in some detail in Section 2.4, there is no need to repeat that information here.

Symbolic execution of the application of a non-primitive function consists of three parts: creation of an entry in a table of potential specializations, addition of an entry onto a stack of pending applications as will be explained in Section 2.4.2, and symbolic execution of the body of the function. Symbolic execution of any application of a non-primitive function first adds an entry to the table of potential specializations for the new potential specialization about to be analyzed. The entry as shown in Figure 2.32 consists of symbolic values for the function and each of its arguments, a slot for a description of the resulting potential specialization, and a slot for storing the symbolic

Function	Arguments	Return Value(s)	Potential Specialization

Figure 2.32: An entry in the table of potential specializations

value(s) for the return value(s)[8]. The symbolic values for the function, arguments, and return values will eventually contain the use annotations characterizing the potential specialization.

Since the use annotations in symbolic values in the new table entry are only intended to reflect information used during symbolic execution of the body of the function, new symbolic values must be created for the table entry. The new symbolic values for the function and the formal parameters have the same values as the ones for the original function and actual parameters, but have distinct use profiles. The new symbolic values are related to the original ones by use dependences that cause all uses of the new symbolic values also to be reflected in the original ones. If new symbolic values were not created for the function and arguments, then the separate uses of a single value made as a result of it being passed to two different functions would be conflated.

Symbolic execution of the function body is initiated using the new symbolic values for the arguments. The expressions that compose the function body are symbolically executed in the fashion being described. The only expression types yet to be addressed are control flow operators. Only conditionals will be discussed since all other control flow operators can be implemented using conditionals.

Symbolic execution of a conditional begins with symbolic execution of the predicate. If the predicate produces a known boolean value, then it is decidable during use analysis which flow of control will be executed by the conditional at runtime. Symbolic execution of the conditional proceeds with either symbolic execution of the consequent or the alternative, as appropriate. If the boolean value of the predicate is not decidable during the analysis, then both the consequent and the alternative must be analyzed separately. The consequent is placed on the *work list* for later symbolic

[8]Symbolic execution of a function may produce one or more return values due to analysis of different potential flows of control.

execution. Symbolic execution of the alternative is initiated immediately.[9]

When symbolic execution produces a return value for the original function application of the program to the input specification (i.e., a result of the program), no analysis remains to be performed in that flow of control. Use analysis takes another flow of control from the work list and initiates symbolic execution of it. If the work list is empty, then eager use analysis has completed. At completion, the entries in the table of potential specializations all contain use annotations characterizing the equivalence classes to which those potential specializations belong.

2.4.2 Termination [10]

Section 2.1 explained that termination of partial evaluation depends on insuring only a finite number of potential specializations are investigated. Section 2.2 proposed a framework in which equivalence classes of applications are utilized in order to guarantee finiteness. By limiting a partial evaluator to investigating at most a finite number of potential specializations for each of a finite number of equivalence classes, termination can be guaranteed. This leaves two goals: determining a finite set of equivalence classes for each function in a program and insuring only a finite number of potential specializations are investigated for each equivalence class.

The equivalence classes to be used in termination of use analysis are based on the use annotations associated with symbolic values. Two function applications belong to the same equivalence class if the use annotations of the applied functions and each of their corresponding arguments are identical. This set of equivalence classes, like those used by many other partial evaluators, is not necessarily finite. In particular, an infinite set of use based equivalence classes may result when use analysis is applied to a nonterminating input program. It is not known in precisely which cases use based equivalence classes are guaranteed to be finite.

[9]It would be completely equivalent to place both the consequent and the alternative on the work list and then take the top element off the work list as the next thread to be analyzed. However, in practice it turns out scheduling either the consequent or the alternative for immediate analysis improves the runtime performance of the analysis. Which of the consequent and alternative is chosen for immediate investigation while the other is placed on the work list appears to be irrelevant, and the order selected is completely arbitrary.

[10]Some portions of this section are taken in whole or part from [31].

Use analysis ensures that at most a finite number of potential specializations are investigated for each equivalence class as follows. Each time symbolic execution reaches a function application, the analysis is about to investigate a new potential specialization. The critical question is whether the new potential specialization about to be investigated would be equivalent to a potential specialization already, or currently being, investigated. A termination mechanism can answer this question in one of three ways: either it believes the new potential specialization would be equivalent to some previously created potential specialization; or it believes the new potential specialization would not be equivalent to any previous potential specialization; or it is uncertain at this time whether the new potential specialization would be equivalent to another.

Previous termination algorithms have not considered the possibility that a termination algorithm might not be able to decide whether a potential specialization is going to be equivalent to some other potential specializations at the point when symbolic execution reaches the application site that initiated the investigation of the new potential specialization, but that the algorithm might be able to give a more definitive answer at some later time. Use analysis differs in this regard. As a result, the structure of a partial evaluator utilizing use analysis as a termination mechanism is somewhat different from most other partial evaluators. Whereas most partial evaluators continue symbolic execution until the termination algorithm indicates symbolic execution ought not to progress any farther, use analysis has more of the flavor of stopping until the termination mechanism indicates symbolic execution ought to proceed. Consequently, use analysis might better be termed a *commencement* mechanism, rather than a termination mechanism.

Furthermore, most partial evaluation algorithms are able to characterize the equivalence class to which a function application belongs immediately upon encountering the application during symbolic execution. Use analysis is unable to associate a function application with an equivalence class until symbolic execution of the function's body has been completed. However, as symbolic execution progresses, more and more information is collected about the DOS of the potential specialization being created. This increased information further and further restricts the size of the equivalence class with which the potential specialization being created will be associated.

Conceptual Implementation

The termination problem in partial evaluation results from recursions.[11] Protecting against divergent analysis is achieved by detecting recursive function applications and then deciding whether to symbolically execute a recursive application or to terminate the analysis of the recursion at that point. (Throughout this explanation, the phrase *terminate a recursion* will be utilized as a short form to designate that symbolic execution of a recursion has been terminated by a decision not to symbolically execute a recursive function application. The operations performed by the symbolic execution engine when a recursion is terminated will be discussed in greater detail later.) The decision to terminate a recursion is made when an initial and a recursive application appear equivalent. Equivalence is determined by comparing the use annotations of the functions and corresponding formal parameters of the two applications to determine whether they are all equivalent.

In order to detect and, as appropriate, terminate recursions, my partial evaluation algorithm maintains a stack of pending function applications for each flow of control. Each time symbolic execution of a function application is initiated, a pending application record is pushed onto the stack. The same record shown in Figure 2.32 on page 41 placed in the table of potential specializations is the one placed on the stack since it contains all of the appropriate information. Each time symbolic execution returns a value from an application, the associated application record is popped off the stack. Conveniently, the record popped off the stack is the one into which the value being returned needs to be recorded; and, a return value is in hand when the record is popped from the stack.[12]

When symbolic execution reaches a function application, use analysis has no information about how the arguments to the function will be used. Consequently, the function application is initially characterized as belonging to the equivalence class

[11] In Scheme, all iteration constructs are implemented as tail recursive functions.

[12] Even when the source language being partial evaluated is properly tail recursive (e.g., Scheme), application records are not removed from the stack until a value is returned from a function. Retaining the records on the stack until a value is returned is mandatory in order for the termination mechanism to be able to detect recursions and prevent divergence. However, introduction of the stack of pending applications means that when utilizing use analysis, symbolic execution of a tail recursive source language is not properly tail recursive.

allowing any element of the domain of values for the function and each of the arguments. When the first recursive call to a function is made, use analysis has only acquired information about the first iteration of the loop. It has only analyzed those uses of information that took place between the initial application of the function and the first recursive application. No information is available regarding how argument values will be used in the second iteration of the loop, so there is no valid basis for deciding whether the first two iterations of the loop are use equivalent. Therefore, symbolic execution continues until a second recursive call is made, at which point information has been acquired regarding uses of information in two iterations of the loop. It is then possible to decide preliminarily equivalence of the first two function applications.

How a termination decision is made using the stack of pending applications is now explained. When a function application is encountered, the stack of pending applications is searched for all applications of closures formed from the same lambda expression. The subset of these applications that meet a condition called *compliance* are then identified. Finally, the compliant applications are compared pairwise, starting with the most recent applications, searching for a pair of pending, compliant applications equivalent to each other. If an equivalent pair is found, the recursion is terminated.

In order for later portions of the analysis to operate correctly, it is important that recursions be terminated at the end of a second equivalent portion of a recursion. The compliance check is designed to ensure the function application about to be performed is the end of the second portion of the recursion. Since the application ending the second equivalent portion will by definition be the beginning of the third equivalent portion of the recursion, the compliance check is used to eliminate from consideration function applications for which the pending application could not possibly represent the beginning of a third equivalent portion of the recursion.

A brief example should serve to motivate the need for the compliance check. Imagine a recursion in which all odd iterations are equivalent to each other and all even iterations are equivalent to each other, but odd and even iterations are not equivalent. When the third application is is reached, there are two applications already on the stack, one odd and one even. The two applications are not equivalent.

When the fourth application is reached, there are three applications on the stack, two odd and one even. The odd applications are equivalent; but, the end of the second equivalent portion of the recursion will not be reached until the fifth application. The compliance test is designed to insure that termination takes place when the fifth, and not the fourth, application is pending.

A value is said to be use compliant with a use annotation if that type of use could correctly be made of that value.[13] For example, `2` is use compliant with `2`, $\perp_{Int}$, and $\perp$, but not with `3`, $\perp_{Bool}$, or `'a`. If the arguments to the pending function application are not compliant with the uses of the formal parameters of an application on the stack, then the pending application could not possibly be the beginning of the next equivalent portion of the recursion for which the application on the stack marked the beginning. This is because the formal parameters of the pending application could never have the same use annotations as the application on the stack.

After decideding whether to terminate a pending application, analysis continues in one of two fashions. If termination is not required, then symbolic execution of the pending application proceeds as previously outlined in Section 2.4.1. If termination is required, then a series of steps are performed to allow the analysis to proceed from the point of the terminated recursion, as explained below. A brief discussion of why two iterations are analyzed before a termination decision is made appears first.

Why Two Iterations are Analyzed

Couldn't a termination decision be made at the time the first recursive application of a function is encountered? Couldn't the values of the arguments of the second application just be compared with the uses of arguments to the first application to ensure they are compliant? The short answer is yes. A termination mechanism based on this rule would be correctness preserving (i.e., would produce correct residual code) and would terminate at least as frequently as the algorithm I advocate. Unfortunately, the algorithm making a termination decision at the first recursive call often yields lower quality residual code than the one I advocate.

[13]Assuming values and uses are represented using the same domains of values, a value is compliant with a use if and only if $use \preceq value$.

Consider once again the example of non-equivalent odd and even iterations presented above. It is quite possible that the arguments of every iteration are compliant with both the uses of the odd and the even iterations, even though the uses of those iterations differ from each other. The algorithm making a termination decision at the first recursive application based on compliance would terminate the recursion after a single iteration in this case and produce a residual recursion with the same structure as the source program; whereas, my algorithm would produce residual code in which the loop has been unrolled one time in order to create different specializations for the odd and even iterations. It is quite possible these two specializations would be more highly optimized than the single, generic version.

One might ask, if two iterations are better than one, why not three, four, or five? It is true that it is always the case that analysis of more iterations might yield a superior result. In fact, the optimal number of iteration to analyze for a given recursion is not in general decidable. However, in practice, it has been found that waiting until at least two iterations have been analyzed so that uses can be compared with each other instead of just compliance with new argument values produces significantly better results. The benefit of setting a minimum number of iterations beyond two appears much, much less significant and does not outweigh the costs of the additional analysis required.

Continuing the Analysis After Terminating a Recursion

When a recursion is terminated, a *use change daemon* is created for each of the equivalent applications. Use change daemons are activated when the use annotations of either the function or one of its arguments in the associated application record changes. Use changes occur due to ongoing symbolic execution performed to create the potential specialization associated with the application record. When a use change daemon is activated, it places an entry on the work list to resume symbolic execution of a recursive function application. The application is the one that was about to be initiated when a termination decision was made. The termination decision was made due to the function application associated with the use change daemon appearing to be equivalent to another application on the stack. When symbolic execution is resumed, the stack is again searched for compliant, equivalent applications.

If equivalent applications are found, execution is immediately reterminated in the manner previous outlined. Otherwise, symbolic execution of the function application proceeds in the normal manner.

When a recursion is terminated, symbolic execution continues from the point where the function application not symbolically executed returns a value to its enclosing expression (i.e., to the continuation of the application). The question of what return value to utilize and how to generate it is left until Section 2.4.3. For now it should just be assumed a value representing the result of the terminated application is somehow created and symbolic execution of the continuation utilizes that value.

Why might it be necessary to resume a recursion due to an incorrectly perceived equivalence? One reason is because every recursive function is composed of two parts: the *head* and the *tail*. The head is the portion of the function performed before the recursive call; and, the tail, the portion performed after. The termination algorithm makes an initial termination decision based only on use information about the heads of recursive functions. In practice, two iterations of a loop might have equivalent heads but distinct tails. Two iterations are not truly equivalent unless both their heads and tails are equivalent.

For example, consider the function in Figure 2.33 that takes a list of numbers as its input and returns a list of the squares of those numbers. The head of each iteration of `square-list-elements` just checks if the end of the list has been reached, and otherwise makes a recursive call to `square-list-elements`. If `square-list-elements` is applied to the list `(1 7 4 12)`, the heads of each of the first two iterations use the information that `lst` is not the empty list, but use no information about the elements of the list. The two iterations therefore appear equivalent. However, the tails of the iterations use the actual values of the list elements `1` and `7` in calculating the squares of those values. When the uses of both the heads and the tails are taken into account, the two iterations are not equivalent. If a specialization created for the first iteration were utilized for the second iteration, the wrong value would be returned by `square-list-elements`. Furthermore, termination of the recursion and production of residual code to compute the return value of `square-list-elements` is far less efficient than creating a specialization of `square-list-elements` just returning the constant value `'(1 49 16 144)`.

```
(define square-list-elements
  (lambda (lst)
    (if (null? lst)
        '()
        (let ((square-rest
                (square-list-elements
                 (cdr lst))))
          (cons
           (square (car lst))
           square-rest)))))
```

Figure 2.33: A function for doubling integers

The problem of equivalent heads but distinct tails is addressed by allowing all termination decisions in a partial evaluator based on use analysis to be retracted at a later time as more information becomes available. Whenever the tails of two iterations that were previously thought to be equivalent are found to be distinct, all termination decisions based on that equivalence are retracted and symbolic execution of the terminated recursion is resumed from the termination point(s). When symbolic execution of an entire program completes, recursive specializations will only have been created for those loops in which there are two iterations for which both the heads and the tails of the iterations were found to be equivalent.

When symbolic execution of a recursion is to be resumed due to retraction of a termination decision, symbolic execution of the continuation of the terminated recursion might already have been initiated. Since symbolic execution of the recursion ought to have continued further, the previously initiated execution of the continuation is no longer needed or desirable. As a result, all flows of control associated with symbolic execution of the continuation must be removed from the work list. Also all data structures created by symbolic execution of the continuation ought to be destroyed. Finally, any entries in any tables of potential specializations created by this flow of control need to be removed. Undoing the results of symbolic execution of the continuation of the terminated recursion turns out to be fairly straightforward, easy, and efficient. The details of the implementation will not be presented here as they are not terribly interesting or illuminating.

No records need to be removed from any stacks of pending applications when a recursion is resumed because a copy of the stack is made when a recursion is terminated. The new copy is used when performing symbolic execution of the continuation of the terminated recursion. If the recursion is later resumed, the original, unadulterated stack is utilized. No other stacks are effected by symbolic execution of the continuation.

2.4.3 Modeling Return Values of Terminated Recursions

When a recursion is terminated, it must be decided how to represent the return value of the application for which symbolic execution is not performed. A return value is required in order to continue analysis of the rest of the program following the terminated application. That is, to perform symbolic execution of the continuation of the application for which symbolic execution is not performed.

Prior to Weise and Ruf [49, 39, 37] all published partial evaluators utilized the completely unknown value, $\perp$, in performing analysis of the continuation.[14] Ruf demonstrated that use of a more precise return value leads to improved optimization during partial evaluation. Symbolic execution is able to perform computations utilizing the more precise return value that would not be possible using the completely unknown return value.

I have adopted a variant of Weise and Ruf's method for computing a more precise representation of the value returned by a terminated recursion. It not only supports the greater potential for optimization sought by Weise and Ruf, but also leads to improved precision of use analysis. The additional computations performed by symbolic execution utilizing the more precise representations of return values produce additional use information. Not only does this generate additional use information about the return values themselves, but also about other values involved in computations performed using the more precise return values.

[14]Weise and Ruf chose their lattice with the opposite orientation of the one I selected, so in his papers this value is referred to as $\top_{Val}$.

```
(define loop
  (lambda (i)
    . . .
    (if (< i 10)
        ( . . .
          (loop . . . )
          . . . )
        . . . )))
```

Figure 2.34: A partial program for which care is required in computing the generalized return value

Weise and Ruf's Method of Modeling Return Values

Conceptually Weise and Ruf's technique for computing representations of return values is to keep track of all of the different values returned by symbolic execution of a function application and generate a single representation encompassing all of the values.[15] The representation of the *generalized return value* is calculated by computing the greatest lower bound of all of the return values utilizing a lattice of the form presented in Figures 2.23 and 2.24 on page 32.

When a recursion is terminated, the iterations that have been symbolically executed are not necessarily representative of all iterations of the recursion that will take place at runtime. The partial program in Figure 2.34 demonstrates how this can be the case. Assume the recursion of `loop` is terminated after iterations in which `i` has taken on successive values of `1` and `2`.[16] For each of these iterations, only the consequent of the conditional has been symbolically executed since all the values of `i` are less than `10`. However, when the recursion is executed at runtime, there is no indication `i` will not take on values greater than or equal to `10`. Utilizing only the return values of all of the existing applications of `loop` to compute the generalized

[15]As explained previously, multiple return values are possible from a single application due to the uncertainty during partial evaluation as to which flow of control will be taken through a function at runtime.

[16]Weise and Ruf's termination mechanism makes the termination decision at the first recursive call, unlike use analysis based termination.

return value might produce an erroneous result since no values produced by the alternative branch of the conditional would be incorporated. For example, the consequent might always return either `#t` or `#f`, which would yield a generalized return value of $\perp_{Bool}$; but, the alternative might return integers so the correct generalized return value is really $\perp_{PEval}$.

In addition, for a partial evaluator like Weise and Ruf's Fuse that produces specialized code as it performs symbolic execution, the specialized code created for the iterations of a recursion symbolically executed up to the point of termination may not be correct for all possible argument values supplied during runtime execution of the recursive specialization. Fuse insures the creation of correct residual code for each specialization and computes a generalized return value correct for all possible return values as follows. When a recursion is terminated, the corresponding arguments of the two equivalent applications are generalized by taking their least upper bound with respect to a use lattice. This produces a generalized argument set. If the generalized arguments are not identical to those in the equivalent applications, then symbolic execution is performed of the function applied to the generalized arguments. When the recursion initiated by this application is terminated, the same process is repeated until the the generalized argument set is the same as those of the most recent equivalent applications. It is the return values of the last application performed using generalized arguments that are utilized in computing the generalized return value.[17]

Applying this principle to the example above, the argument values `1` and `2` are generalized to produce a new argument of $\perp_{Int}$. `loop` is then applied to $\perp_{Int}$. If the new recursion initiated by `(loop` $\perp_{Int}$`)` is terminated with an identical application of `loop` to $\perp_{Int}$, then the generalized return value is computed by generalizing the return values of the initial application of `loop` to $\perp_{Int}$. However, if the recursion initiated with `(loop` $\perp_{Int}$`)`, is terminated with an application of `loop` to `#t`, then the process must be repeated since the generalized argument previously utilized, $\perp_{Int}$, does not include the new type of argument utilized in the terminated recursion.

[17]Weise and Ruf consider the process of repeatedly generalizing argument sets and performing symbolic execution of the generalized applications to be part of their termination mechanism as opposed to part of the process of generating a return value approximation due to their approach to code generation.

Generalizing $\perp_{Int}$ and `#t` produces $\perp_{PEval}$. Symbolic execution is now performed of `(loop` $\perp_{PEval}$`)`. Assuming this recursion is terminated with another application of `loop` to $\perp_{PEval}$, the process is now completed and the generalized return value can be computed by generalizing the return values of the initial application of `loop` to $\perp_{PEval}$.

There is one complicating factor in computing generalized return values. Symbolic execution of the continuation of a terminated recursion utilizing a generalized return value yields at least one new return value for the equivalent recursive application. Recomputation of the generalized return value utilizing all of the return values including the newest one might yield a different value than the one previously computed. In this case, symbolic execution of the continuation of the terminated application must be performed again using the new generalized return value. This process is repeated until the resulting generalized return value ceases to change yielding a fixed point.

I now provide an example. Figure 2.35 contains a simple length function borrowed from Ruf [37]. Assume `length` is specialized on the value $\perp$. Since the value of `(null? lst)` is not decidable during symbolic execution, both the consequent and the alternative of the conditional are analyzed. The consequent returns the value `0`. The alternative makes a recursive call to length. Assuming the recursion is terminated, a return value is needed to proceed with symbolic execution of the terminated application. Since only the single value `0` has been returned by the first application of `length`, the initial generalized return value is `0`. The continuation of the terminated application adds `1` to the initial generalized return value yielding a second return value of `1`. Addition of the new return value to the application record causes a new generalized return value to be computed. Using the lattices of Figures 2.23 and 2.24 on page 32, the greatest lower bound of `0` and `1` is $\perp_{Int}$. Since the new generalized return value differs from the previous one, symbolic execution of the continuation of the terminated application is performed again utilizing the new generalized return value. The addition of `1` and $\perp_{Int}$ yields the value $\perp_{Int}$ as a new return value for `length`. A recomputation of the generalized return value utilizing the three values `0`, `1`, and $\perp_{Int}$ once again produces $\perp_{Int}$, so a fixed point has been reached and no further computation is required in order to determine the correct generalized return value for `length` applied to $\perp$.

```
(define length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst))))))
```

Figure 2.35: A function for computing the length of a list

Modeling Return Values in Use Analysis

Application of Weise and Ruf's approach for modeling return values to use analysis can be demonstrated by considering once again the example in Figure 2.34. For use analysis, the recursion is terminated after symbolic execution of two iterations of the recursion. The applications of `loop` to both `1` and `2` have been symbolically executed and in this case the application of `loop` to `3` is the one that is terminated. At that point the stack of pending applications is as shown in Figure 2.36. Taking the greatest lower bound of the argument values of `1`, `2`, and `3` using the lattice in Figures 2.23 and 2.24 produces a generalized argument of $\bot_{Int}$. Symbolic execution proceeds with the application `(loop` $\bot_{Int}$`)`. For the sake of this example, I will assume successive iterations of `loop` are initiated by recursive applications to the identical argument value, $\bot_{Int}$. When the new recursion is terminated, the argument values would once again be generalized to produce a new set of generalized arguments. Since the same argument value of $\bot_{Int}$ is used in every iteration, the new generalized argument is $\bot_{Int}$, which is identical to that for the earlier iteration. Consequently, the generalized return value is computed by taking the greatest lower bound of the return values of the first application of `loop` to $\bot_{Int}$.

My algorithm operates the same as Weise and Ruf's with two small additions. In Weise and Ruf's partial evaluator in which termination is not based on use analysis, the generalized return value can be created by repeatedly generalizing arguments and performing new applications until the process converges. The generalized return value is the generalization of the return values of the final recursion that is analyzed. Analysis of the tails of the iterations of the intermediate recursions is not necessary. However, when termination is based on use analysis, correct termination of each of

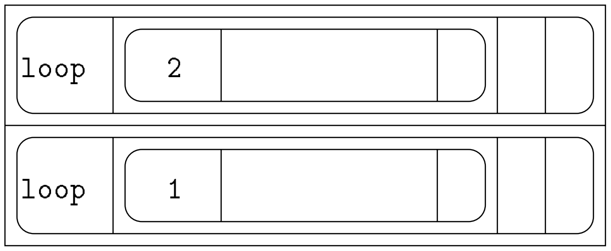

Figure 2.36: Stack at time of first termination

the intermediate recursions can be dependent on analysis that is performed of the tails of those intermediate iterations. In order to get use information about the tails of the iterations, symbolic execution of the tails must be performed. The purpose of the first addition to Weise and Ruf's algorithm is to ensure symbolic execution of the tails.

Symbolic execution of the tails of intermediate iterations is facilitated by utilizing the continuation of each terminated application as the continuation of the application of that function to a generalized argument set. When symbolic execution of the application of the function to the generalized argument set is completed, symbolic execution proceeds with the continuation assigned to that application. Since the continuation is that of the terminated application, symbolic execution proceeds with the tail of the earlier iteration.

Continuing the `loop` example from Figure 2.34 on page 51, when the second recursion is terminated, the stack appears as shown in Figure 2.37. (The dotted lines designate the generalization of the arguments. The argument to the terminated application that has not been performed appears on the right.) When the first application of `(loop` $\perp_{Int}$`)` returns, its return value is supplied to the continuation of the application of `loop` to `3`.

The second addition to Weise and Ruf's approach is needed in order to ensure that uses of generalized values imply use of information about the values utilized in creating the generalized values. A link in the use dependence graph between generalized values and the values used in computing the generalization is required for correct use annotations to be created for all the other values in a use dependence graph. Otherwise, there would be no use link between the value returned by a recursive

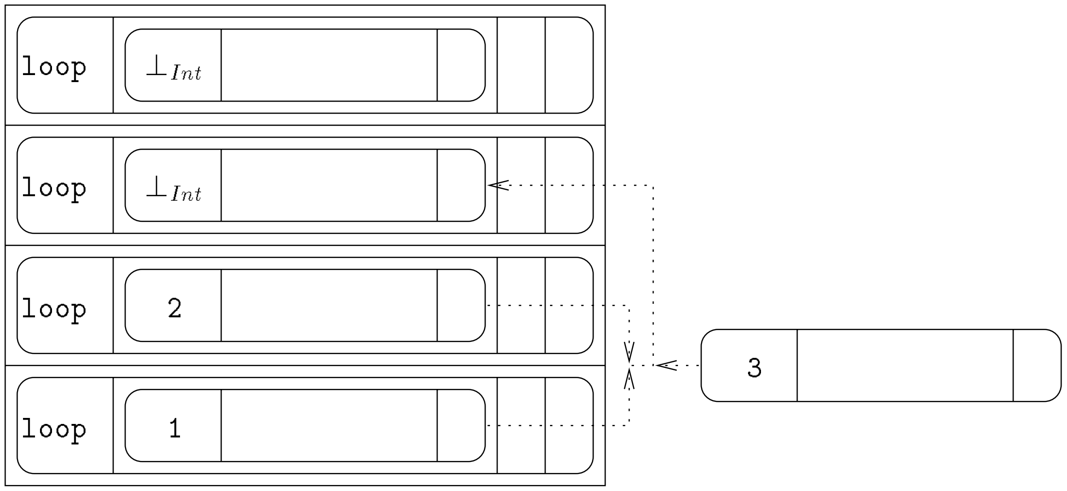

Figure 2.37: Stack at time of second termination

function and the arguments supplied to initiate the recursion [37].

When two values are generalized to produce a new value, two use dependences must be created. Each use dependence links use of the new generalized value to use of one of the two values combined to form the generalization. Any use made of the generalized value is implied for both of the values used in computing its value. Similarly, if more than two values are generalized, then a use dependence is created from the generalized result to each of the values used in computing the generalized result.

Figure 2.38 is an updated version of Figure 2.37 with the use dependences created by generalization of arguments included. Assuming the first iteration of `(loop` $\perp_{Int}$`)` returns two values, `#t` and `#f`, Figure 2.39 shows the stack at the point at which the tail of `(loop 2)` is about to be executed. The `(loop` $\perp_{Int}$`)` application is just about to be popped off the stack. $\perp_{Bool}$ is the generalized return value that will be supplied to the continuation that evaluates the tail of `(loop` $\perp_{Int}$`)`.

The use dependences created by the generalization operation are necessary to insure appropriate use is recorded for all of the values utilized in each iteration of a recursion. Without these use dependences, use of information about the generalized return value could not imply use of information about argument values utilized in computing the generalized return value.

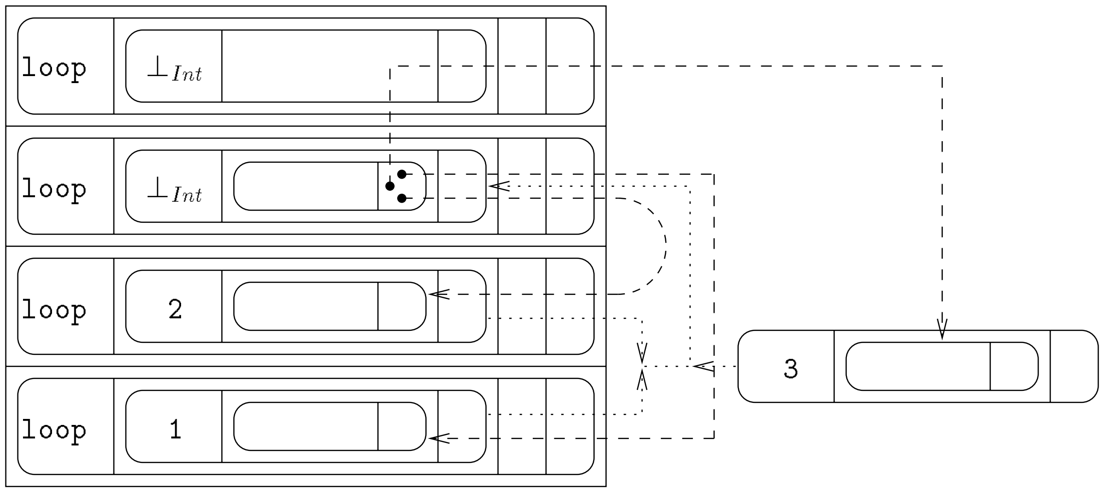

Figure 2.38: Stack at time of second termination including use dependences

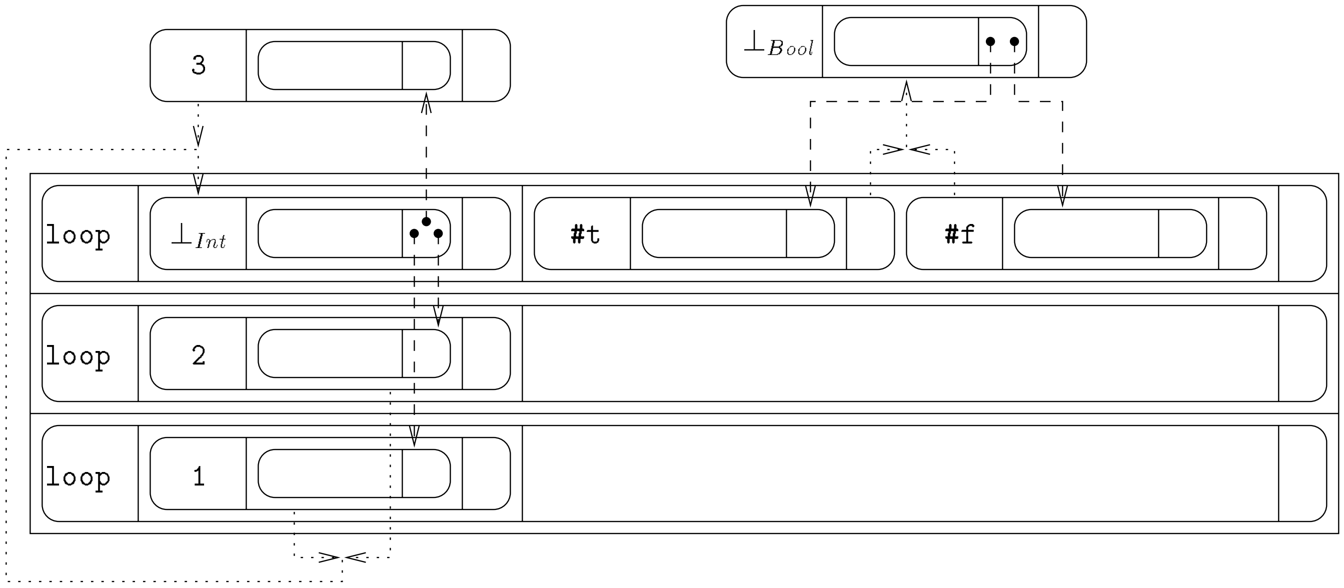

Figure 2.39: Stack just before the tail of `(loop 2)` is executed

2.5 Shortcomings of Eager Analysis

Eager use analysis records information utilized in performing symbolic execution of expressions. As such, it is an approximation to the CF-DOS. It does not attempt to determine the minimum amount of information necessary to generate the same result. Consequently, the characterization of an expression produced by eager use analysis often is an overspecification of the information required to produce a given specialization. For example, in the expression `(integer? (+ 1 2))`, `+` uses the identity of both of its arguments to produce the result `3`. Eager use analysis modifies the use annotations of the arguments `1` and `2` to reflect use of the values of the arguments. The function `integer?` only uses that `3` is an integer, not its value. `integer?` would return the same result for any two integer arguments to which `+` were applied. A specialization of the expression `(integer? (+ 1 2))` really only depends upon the types of `1` and `2`, not on their identities (values). Embedding `(integer? (+ 1 2))` in an even larger expression leads to yet greater overspecification of the information utilized. In the expression `((lambda (a b) a) #f (integer? (+ 1 2)))`, no information about `1` or `2` is used since the result of `integer?` is thrown away; however, the use annotations resulting from eager use analysis still indicate that the identities of the integers are used.

The overspecification of use inherent in eager use analysis yields smaller equivalence classes than are implied by the semantics of a program. This can cause a partial evaluator whose termination mechanism is based on eager use analysis to diverge, as previously demonstrated in Section 2.2.2 using the "counting up" factorial example. Similarly, failure to recognize two applications are equivalent due to overspecification of use can cause the creation of duplicate potential specializations and potentially unneeded duplicate residual code.

For example, consider the strange version of factorial appearing in Figure 2.40. The extra parameter `i` is unneeded, but a computation is performed utilizing it during each iteration of the recursion. Eager use analysis of `strange-fact` applied to an unknown value of `n` and the value `1` for `i`, yields a nonterminating recursion. This is because in every iteration of the recursion a different value of `i` is used in computing the value of `i` for the next iteration. As will be explained in the Section 2.6, lazy use

```
(define strange-fact
  (lambda (n i)
    (if (zero? n)
        1
        (* n
           (strange-fact (-1+ n) (* i 2))))))
```

Figure 2.40: A strange version of factorial with an unnecessary extra argument

analysis does not record different uses of `i` for each iteration and therefore terminates.

Finally, the overspecification of use inherent in the eagerness of eager use analysis ought to be distinguished from the over or underspecification of use resulting from the inability of a given lattice of uses to precisely capture the uses of information made by some primitives. The former results from certain primitives not using all the information available about all of their arguments. The latter results from the necessity to approximate some uses of information.

2.6 Lazy Use Analysis

Eager use analysis is based on recording information utilized in performing delta reductions, even those that do not contribute to the result of a program. Equivalence classes based on information used in performing intermediate computations not contributing to the result of a program are overly small and lead to more frequent divergence since they unnecessarily distinguish between potential specializations with identical input/output behavior.

Equivalence classes based on eager use analysis fail to take the context of expressions into account. Lazy use analysis differs by utilizing context information. Because of the utilization of context information, lazy use analysis is an approximation to the CS-DOS.

Information can contribute to a result of a program in one of two ways, either by affecting the data or the control flow of a program. The means in which information about a value percolates through the data flow of a program to affect the final result is fairly straightforward and is evident from the dynamic data flow graph of the execution of a program. Values can also affect the result of a program when they are used in making a control flow decision (e.g., by the predicate of a conditional). When a control flow decision effects the computation of the final result of a program, the information used in making the control flow decision has been used in generating the program's result.

Whereas eager use analysis records information as being used as soon as a delta reduction is performed, lazy use analysis must trace the flow of information through an entire program before it can precisely establish what uses of information are salient. Lazy use analysis of a delta reduction only produces use dependences between symbolic values, it does not record any actual uses of information. How a graph of use dependences are utilized in order to generate use annotations for symbolic values will be explained after a discussion of how the use dependences are created for lazy use analysis. Greater detail on the low level implementation of these concepts, including the set of uses and use dependences utilized, appears in Chapter 3.

2.6.1 Creating the Graph of Use Dependences

This section discusses how arcs and nodes are created for several different types of expressions in the use dependence graph. Delta reductions are presented first. Then control flow operators are discussed. Amongst control flow operators, the two types outlined are conditionals and function applications.

Each time a delta reduction is performed, a symbolic value is produced for the result and use dependences are created between the result and each of the arguments to the delta reduction. The use dependences express how use of information about the result implies use of some corresponding information about the arguments. For example, performing the delta reduction `(+ 1 2)` yields the result `3`. When utilizing the domains in Figure 2.22 on page 31, repeated as Figure 2.41, and the lattice in Figures 2.23 and 2.24 on page 32, repeated as Figures 2.42 and 2.43, the use dependences created by the application of `+` express the following relationships. Use of the identity of the integer result would imply use of the identities of both of the arguments. However, use of the integer property of the result would only imply use of the integer properties of the arguments. Finally, the need to compute the result would

Int	$=$	$\mathtt{0} + \pm\mathtt{1} + \pm\mathtt{2} + \cdots$	integers
		$\bot_{Int}$	unspecified integer
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
		$\bot_{Bool}$	unspecified boolean
Sym	$=$	$\mathtt{'a} + \mathtt{'b} + \cdots$	symbols
		$\bot_{Sym}$	unspecified symbol
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\bot_{Pair} \equiv \bot_{PEval} \times \bot_{PEval}$	unspecified pair
$Closure$	$=$	$Lambda \times Env$	closure values
		$\bot_{Clos}$	unspecified closure
Env	$=$	$(Id \rightarrow PEval)^{\star}$	environments
$Kval$	$=$	$Int + Bool + Nil + Pair + Closure$	known values
$Bots$	$=$	$\bot_{Int} + \bot_{Bool} + \bot_{Pair} + \bot_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \bot_{PEval}$	partial evaluation values
		$\bot_{PEval}$	unspecified value

Figure 2.41: Value domains for partial evaluation of a pure subset of Scheme (repeat of Figure 2.22)

imply the need to compute both of the arguments. However, the use annotations of the arguments 1 and 2 are not immediately affected by symbolic execution of the addition primitive. This differs from eager use analysis in which the use annotation of 1 would immediately be changed to 1 and the use annotation of 2 would immediately be changed to 2.

Figure 2.44 shows the application record, symbolic values, and use dependences built for the `((lambda (a b) a) #f (integer? (+ 1 2)))` example in Section 2.5. The dependences between 3 and both 1 and 2 are labeled as *id* for identity. As explained in the previous paragraph, these dependences express that use of the identity of the former implies use of the identity of the latter; use of the type of the former, use of the type of the latter; and, the need to compute the former, the need to compute the latter. The `value->type` dependence between `#t` and 3 results from the `integer?` application. It expresses that use of the boolean value of `#t` implies use of the type (i.e., integerness) of three. Furthermore, the need to compute `#t` implies the need to compute 3. However, use of the boolean property of `#t` implies no use of any information about 3 since all values returned by `integer?` are booleans. By

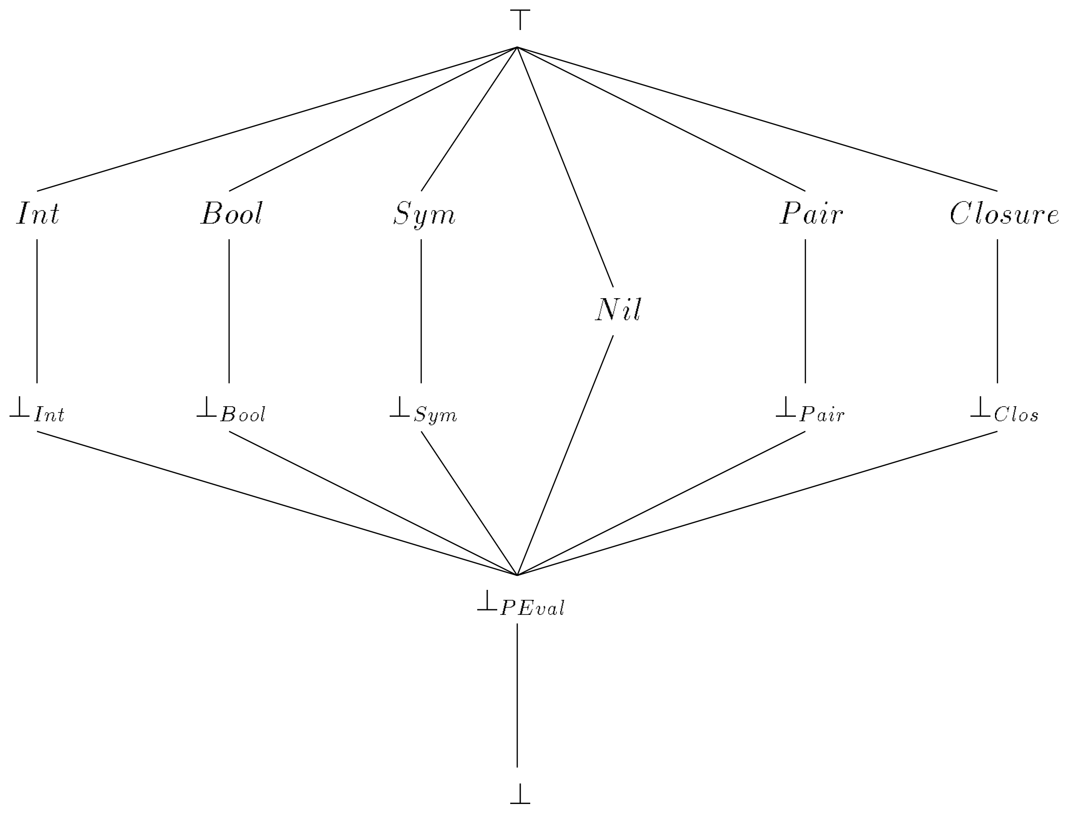

Figure 2.42: Information lattice formed using partial evaluation value domains (repeat of Figure 2.23)

$$\forall < x, y >, < x', y' > \in Pair.((< x', y' > \preceq < x, y >) \leftrightarrow ((x' \preceq x) \wedge (y' \preceq y)))$$
$$\forall < l, e >, < l', e' > \in Closure.((< l', e' > \preceq < l, e >) \leftrightarrow ((l' = l) \wedge (e' \preceq e)))$$
$$\forall e, e' \in Env.((e' \preceq e) \leftrightarrow (\forall < i, s' > \in e'.(\exists < i, s > \in e.(s' \preceq s))))$$

Figure 2.43: Information lattice equations to supplement the lattice diagram (repeat of Figure 2.24)

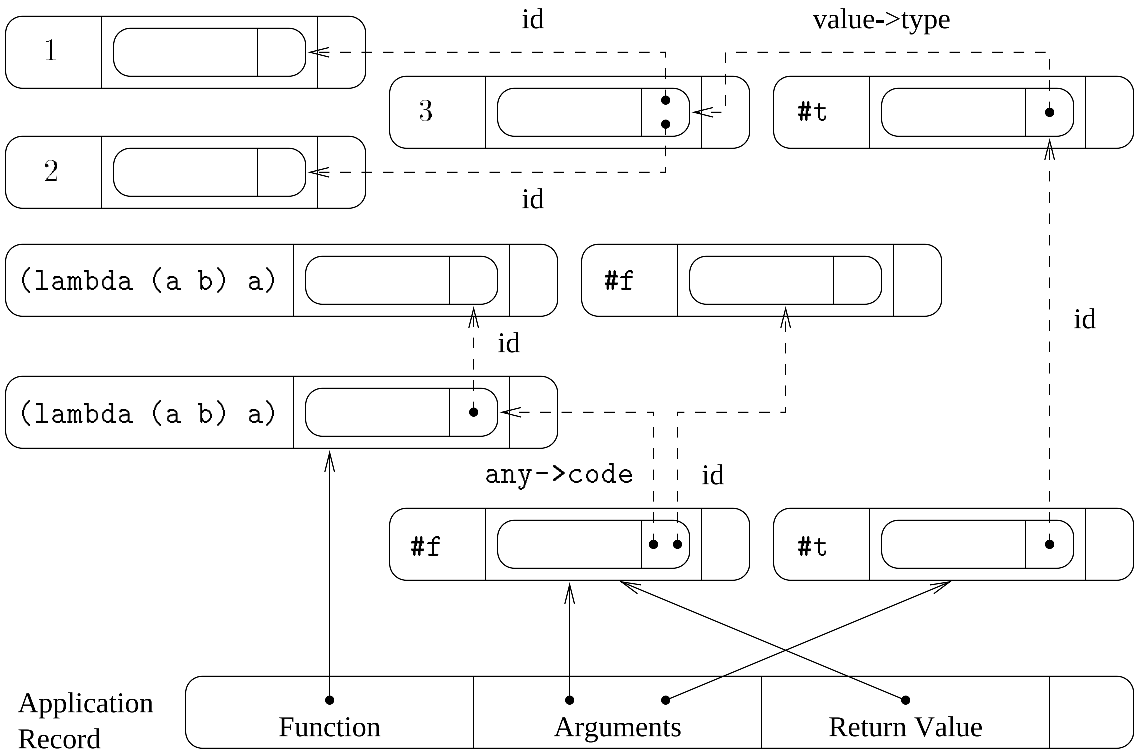

Figure 2.44: The application record, symbolic values, and use dependences built for `((lambda (a b) a) #f (integer? (+ 1 2)))`

transitivity, all uses of `3` implied by uses of `#t` are also implied of `1` and `2`.

Looking at the application record in Figure 2.44, the function is the closure for `(lambda (a b) a)`. The arguments are the `#f` explicit in the expression `((lambda (a b) a) #f (integer? (+ 1 2)))` and the `#t` resulting from the computation `(integer? (+ 1 2))`. As presented for eager use analysis, a copy is made of the function and each of the arguments when creating the application record so there is a distinct place for recording uses of information resulting from symbolic execution of the body of the applied function. As the semantics of copying would imply, an *id* use dependence is created between each of the copies and the original. The only value returned by the application in the example is the first argument, `#f`. Note, there is no use dependence chain from the result to any of the values `#t`, `3`, `2`, or `1`. Regardless of what use is made of the result of the expression, no uses will ever be implied for any of the values `#t`, `3`, `2`, or `1`. The meaning of the `any->code` dependence between

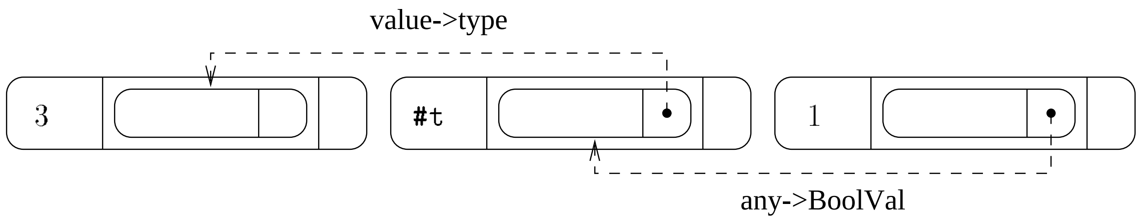

Figure 2.45: The symbolic values resulting from `(if (integer? 3) 1 2)`

`#f` and the closure `(lambda (a b) a)` will be presented later.

Symbolic execution of dynamic control flow operators creates use dependences. The return value of a conditional depends upon whether the consequent or the alternative of the conditional is evaluated. Any use of information about the result of a conditional is dependent on the boolean value of the predicate that determined which branch of the conditional was executed. As a result, symbolic execution of a conditional creates a use dependence between the symbolic value for the result of the conditional and the symbolic value for the predicate.[18]

Figure 2.45 shows the symbolic values for the expression `(if (integer? 3) 1 2)`. The `any->BoolVal` dependence between the symbolic values for `1` and `#t` signifies that any use of the result of the conditional, `1`, implies use of the value of the boolean `#t` that caused the consequent to be executed. By transitivity, use of any information about the result also implies use of the integerness of `3`.

A function application is a dynamic control flow operation similar to a conditional. The value returned by a function application is not only dependent on the arguments to which the closure is applied, but also on the closure being applied. Since a closure is composed of both a lambda expression and an environment, the return value of an application may be dependent on three different types of information: the argument values, the lambda expression evaluated[19], and the values stored in the environment of the closure. Separate use dependences may be needed to capture each of these

[18]If the boolean value of a predicate is not decidable during symbolic execution, both branches of the conditional must be analyzed. In this case, no dependence is needed between the two possible results and the predicate since the boolean value of the predicate was unknown so no control flow decision was made during partial evaluation.

[19]Lazy use analysis defines code equivalence in terms of identical lambda expressions. Of course, function equivalence is in general undecidable.

three types of information utilization.

The rules for symbolic execution of delta reductions and control flow operators other than function applications naturally handle the dependences between the value returned by a function application and the values of the arguments and those in the environment of the closure.[20] A new type of dependence is needed to capture the dependence of the return value on the code portion of a closure. This dependence between the value returned by an application and the closure applied represents the control flow dependence of the the value returned on the lambda expression from which the closure was formed. The `any->code` dependence between the symbolic value for the return value of `#f` and the closure for `(lambda (a b) a)` in the previously discussed example in Figure 2.44 is an example of this new type of dependence.

In the absence of the `any->code` dependence, the following two applications would yield the identical set of symbolic values and use dependences: `(+` $\perp_{Int}$ $\perp_{Int}$`)` and `(*` $\perp_{Int}$ $\perp_{Int}$`)`. Since all the values and use dependences would be identical, the two applications would be placed in the same equivalence class. This is clearly not appropriate as the two functions applied are clearly not equivalent and neither are the potential specializations that would result from the applications presented. The `any->code` dependences codify that the results of the two applications each depend on the code of the closures applied, so if any information is utilized about the results, then the applications are not equivalent.

Embedding the conditional example from Figure 2.45 in a closure yields the expression `(lambda (a) (if (integer? a) 1 2))`. Specializing that closure on the input value `3` produces the result in Figure 2.46. The example now demonstrates two types of control flow dependences: one for a function application (`any->code`) and one for a conditional (`any->BoolVal`).

2.6.2 Generating Use Annotations

So far it has been explained how symbolic execution creates a use dependence graph for lazy use analysis. The graph represents how use of information about one symbolic

[20]Values in the environment of the closure can naturally be thought of as extra arguments in the application, which is the implementation technique used for lazy use analysis.

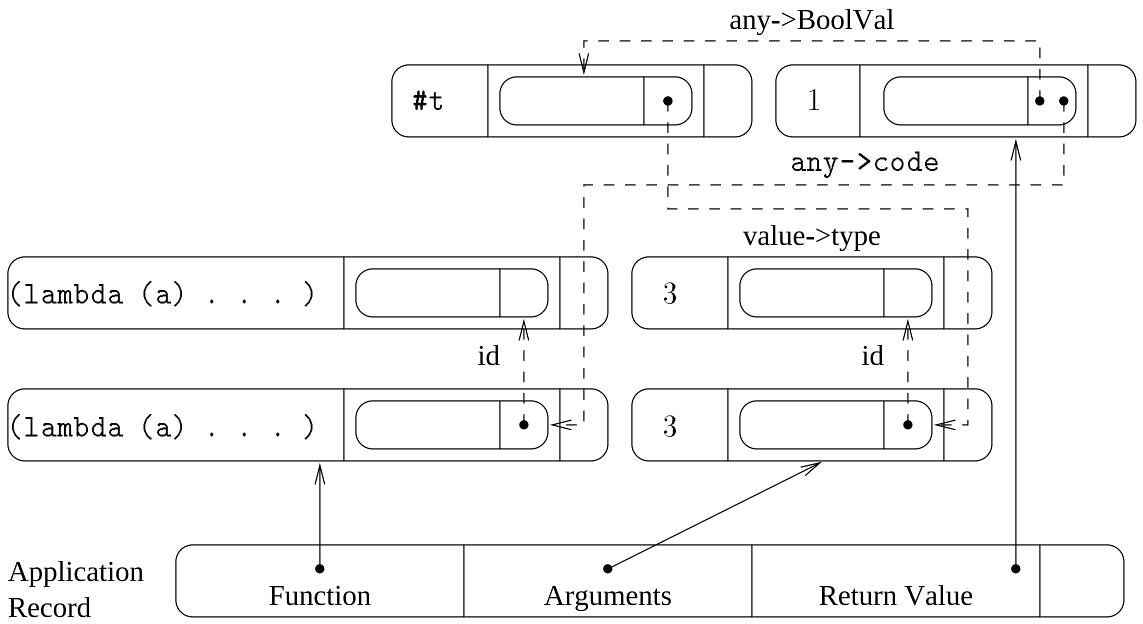

Figure 2.46: The symbolic values and application record resulting from `((lambda (a) (if (integer? a) 1 2) 3)`

value implies use of information about possibly many other symbolic values. However, it has yet to be explained how use is ever asserted about any value, causing use information to start being propagated along the use dependence arcs.

Based on the assumption that the purpose of executing a program is to compute its result, lazy use analysis was designed to record the information used in computing the result(s) of symbolic execution of a program.[21] It is not surprising, therefore, that the propagation of use information over a use dependence graph is initiated by asserting some use of the return value(s) of a program. The critical question is the type of use to assert.

The analysis could assert that all the information present in each return value is needed/used. For example, if the value `3` were one of the family of return values, use of its 3-ness would be asserted. If another return value were the unspecified boolean value, $\perp_{Bool}$, $\perp_{Bool}$ use would be asserted of that return value. If the return value

[21]Since symbolic execution is based on partial information, there may be many possible flows of control so there may be a family of possible return values.

were an aggregate, use of the value of each of the components of the aggregate would be asserted.

Asserting all of the information present in all results is needed is potentially an overspecification of the actual use. Much as it was observed when discussing eager versus lazy use that not all information produced by a subexpression may be utilized, not all the information present in all the return values of a program may be important. For example, when partial evaluation produces a family of possible return values, not all of the return values would be produced by any single execution of the input program. Asserting all the information in all of the return values is utilized can lead to an overspecification of the information used to produce the result of one single execution.

The overspecification of use inherent in asserting all of the information about all results is used can yield an analysis based on artificially small equivalence classes. As previously discussed, such analyses are not particularly well suited for termination decisions since they lead more frequently to divergence, but are well suited for making reuse decisions about potential specializations that are correctness preserving. Consequently, one possible approach for creating a use analysis for making code reuse decisions, but not for termination, has been identified. Selecting what to assert to produce a use analysis for making termination decisions is subtler.

The use of some information about results must be asserted or no use annotations will be produced for any of the values in a program. To ensure an underspecifying analysis, the use asserted about results must be guaranteed to be the minimal amount ever used. It safely can be asserted that any program must produce its result, even if no information about the result is used. The question is how to represent this concept as a use.

Intuitively, there is less information content in needing to produce a value than in needing any information about that value. On the other hand, needing to produce a result represents a requirement for slightly more information about the result than not requiring any information about the result at all. These two intuitive information bounds dictate the placement within an information lattice of the use annotation representing the need to compute a result.

A discussion of the difference between $\perp_{PEval}$ and $\perp$ was postponed during an earlier presentation of the lattice repeated in Figures 2.42 and 2.43 on page 62. $\perp$ represents that absolutely no information about a value has been used; whereas, $\perp_{PEval}$ represents that the value must be computed, but no information beyond the need for the value to be computed is used. Based on these definitions, lazy use analysis initializes all symbolic values with a use annotation of $\perp$. An underspecifying use analysis for termination asserts the result(s) of a program must be computed by assigning a use of $\perp_{PEval}$ to the return value(s) of a program. Based on those use assignments and the use dependences, assigning uses to all symbolic values produced during symbolic execution is straightforward.

The last piece of the puzzle is to explain how $\perp_{PEval}$ is handled by the different types of use dependences. The need for a primitive function to produce a result implies the need for it to be supplied with a value for each argument in which the function is strict. Consequently, a use dependence arc is created between the symbolic value for the result of a primitive and the symbolic values for each of its strict arguments, causing $\perp_{PEval}$ use of the result to imply $\perp_{PEval}$ use of the strict argument(s). Conditional control flow operators produce use dependences that convert the need for the conditional control flow operator to produce a result into use of the information required to decide to execute the flow of control that produces the result. For example, $\perp_{PEval}$ use of the result of a conditional implies use of the boolean value of the predicate that caused the branch of the conditional producing the result to be executed.

Figure 2.47 is an updated version of Figure 2.46 created by asserting the result of the program `(lambda (a) (if (integer? a) 1 2))` specialized on `3` must be computed. The process of assigning uses is initiated by asserting $\perp_{PEval}$ use of the return value, `1`. As a result of this assertion and the `any->BoolVal` dependence, `#t` use is asserted of `#t`. This in turn causes $\perp_{Int}$ use to be asserted for the copy and original argument, `3`. The `any->code` dependence and the assertion of $\perp_{PEval}$ use of the return value causes use of the code portion of the original and copy of the function to be asserted.

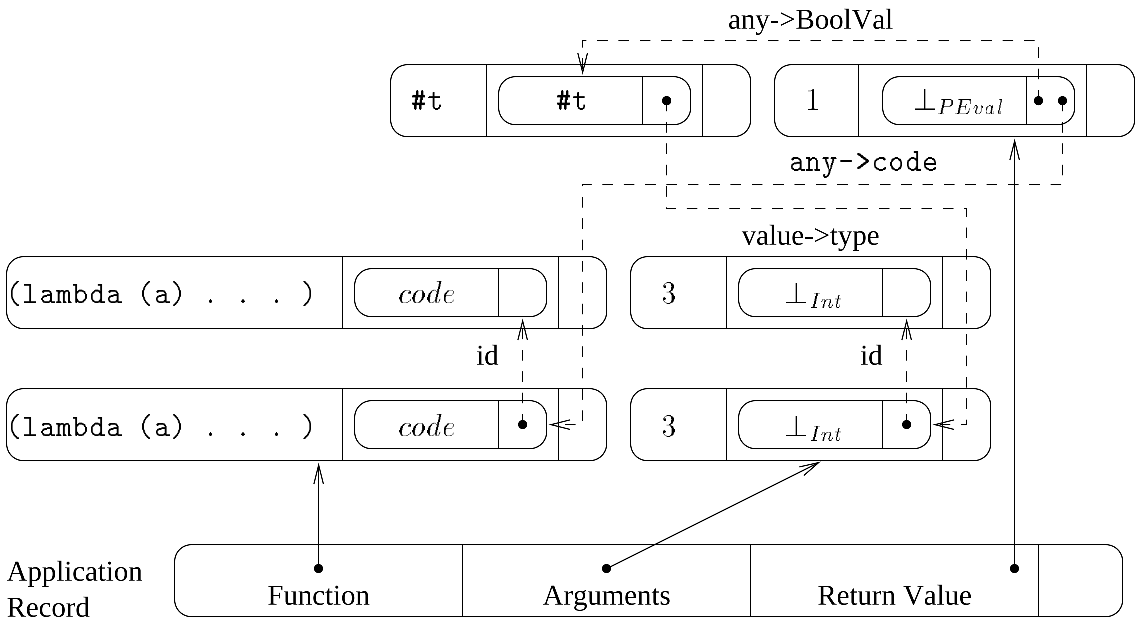

Figure 2.47: Use annotations for `((lambda (a) (if (integer? a) 1 2) 3)`

2.6.3 Termination

A partial evaluator based on lazy use analysis is unable to collect any information about uses before the question whether two iterations of a loop are equivalent is first asked. Why? Because, no information about uses is propagated through a use dependence graph until a return value of a program is generated and some use of the return value is asserted. Assuming the return values of a program are dependent on the return value of the recursion about which a termination decision is being made, the return values of the program could not possibly be generated before the return value of the recursion. However, the return value of the recursion cannot be generated until either symbolic execution of the loop runs to completion or until the loop is terminated and a generalized return value is created to represent the return value of the entire recursion.

The solution to this chicken and egg problem with lazy use analysis is the same one eager use analysis utilizes to address the problem that the tails of iterations have not yet been analyzed when a termination decision first needs to be made. A preliminary termination decision is made that might be retracted later if it proves to have been incorrect. Since there is no use information available when lazy use analysis first considers whether two iterations of a recursion are equivalent, all applications initially fall within the same equivalence class: the one for no utilization of information about any values. As a result, all iterations of each recursion are initially deemed equivalent, and symbolic execution of all recursions is terminated after two iterations. Later, information propagation through the use dependence graph may invalidate equivalence decisions causing symbolic execution of the corresponding recursions to be reinitiated.

There exists one subtlety in deciding to resume symbolic execution of a recursion when two iterations of the recursion become nonequivalent. As use information propagates through a use dependence graph, it arrives at some nodes before others. It must be ensured that any non-equivalent use annotations are not the result of information having reached some nodes of the use dependence graph before others. In other words, the check for equivalence must only be performed after the effects of changing some use annotations have been fully propagated throughout a graph.

For example, imagine a recursive function of a single argument whose value is `3` at the first function application, `4` at the first recursive application, and `5` at the next recursive application. Symbolic execution is terminated at the second recursive application due to both the `3` and the `4` being annotated as unused at the point at which the termination decision is first made. However, the integer property of each of the argument values might be contributing to the return value of the program. Eventually use information might propagate through the use dependence graph to change the use annotations of all of the arguments to be $\perp_{Int}$. But, an intermediate state might exist in which the use annotation of one of the arguments is $\perp_{Int}$ and the use annotation of the other is $\perp$. As a result, it is critical the decision to resume symbolic execution of a previously terminated recursion only be made when propagation of use information through a use dependence graph has quiesced; otherwise, equivalent iterations might appear non-equivalent, leading to divergence. This issue is discussed in greater detail in Section 3.2.2.

2.7 Approximating Use

In theory there is a precise specification of the information needed about all of the values referenced by any program in order to compute its result. However, in practice it is almost never possible for any real partial evaluator to represent precisely the information used. As a result, any realizable use analysis must make approximations when it represents the information about values used. When use analysis must approximate the information utilized and the implications of those approximations is now presented.

Two types of approximation of use are discussed in this section. The first results from any form of use analysis being an approximation to the fundamental information used by two potential specializations. While lazy use analysis comes closer than eager use analysis to representing the fundamental information utilized, both are approximations to the ideal. Second, no set of domains and use lattice can precisely represent the types of information utilized in executing all of the primitives of any interesting programming language.

The choice of how and when to record use is one source of approximation in any use analysis. Eager use analysis systematically overspecifies the information used about values. This results from recording the information used in performing all delta reductions and control flow decisions, as opposed to recording only the information needed to compute the result. Lazy use analysis lessens, but does not necessarily eliminate, the overspecification of use by attempting to capture only those uses contributing to the result. Overspecification remains from the assumptions that the result needs to be computed, the return value of a conditional depends on which branch is taken, and the return value of a function application depends on the code associated with the closure applied.

The inability to precisely represent the information used by some delta reductions causes approximation. As previously presented, it is not possible to represent precisely the information used in performing the reduction `(< 3 5)` utilizing the information domains in Figure 2.22 on page 31. An approximation either over or under specifying the actual use is necessary.

A fundamental decision in implementing either eager or lazy use analysis is whether to over or under specify use for those delta reductions for which a precise representation is not possible. Four different analyses result from the choices of eager versus lazy analysis and over versus under specification. Analyses based on overspecification are most useful for deciding when a specialization can be reused for a new set of values since the approximations have been made in a direction ensuring reuse is correctness preserving. Analyses that underspecify use are probably most desirable for making termination decisions since they minimize the probability of divergence. However, termination based on underspecification of use tends to reduce the number of specializations generated. This can lead to lower quality residual code due to partial evaluation failing to create some useful specializations. An implementor might choose to allow a partial evaluator to diverge more often in order to yield improved residual code in those cases in which the partial evaluation terminates. This fact notwithstanding, my presumption in the following discussion is that overspecification will be used for reuse decisions and underspecification for termination decisions. Retaining both types of use information requires performing two types of analysis in parallel, with one being utilized to terminate both itself and the other one.

The diagram in Figure 2.48 represents the relative amounts of information utilization that might be recorded by different types of analyses for the same value. Points higher on the diagram should be interpreted as representing use of a greater amount of information. The dashed lines represent that the lower value in the diagram is related to the upper value by the previously defined $\preceq$ operator read as *has less than or the same amount of information as.*[22] While the diagram is not a lattice, it can be interpreted in much the same manner. $Value$ is the value whose use is being recorded. EU_{reuse} is the result of eager use analysis using overspecification when no precise representation is possible; EU_{term}, the result of eager use analysis with underspecification; LU_{reuse}, the result of lazy use analysis and overspecification; and, LU_{term}, the result of lazy use analysis and underspecification.

The relative positions of the different analyses in the diagram in Figure 2.48 arise as follows. Any correct use analysis always records use of some subset of the information available about a value. It is not possible to use more information about a value

[22]Figure 2.48 is based on the presumption the same domains are being used to represent both values and uses so the values are comparable using the $\preceq$ operator.

$$\begin{tikzcd}
 & Value \arrow[d, dashed, no head] & \\
 & EU_{reuse} \arrow[dl, dashed, no head] \arrow[dr, dashed, no head] & \\
LU_{reuse} \arrow[dr, dashed, no head] & & EU_{term} \arrow[dl, dashed, no head] \\
 & LU_{term} \arrow[d, dashed, no head] & \\
 & \perp &
\end{tikzcd}$$

Figure 2.48: Relative amount of use information recorded by different analyses

than it contains. An underspecifying use analysis always records less information utilization than a corresponding overspecifying analysis. Consequently, the analysis with the *term* subscripts always appear below the corresponding analysis with *reuse* subscripts. By definition, lazy analyses always record use of less information than corresponding eager analysis. Finally, LU_{reuse} and EU_{term} are incomparable. Lazy analyses in general record less information utilization; however, LU_{reuse} is a lazy analysis based on overspecification of use in some cases while EU_{term} is an analysis based on an underspecifying approximation. The net result is either of the two analyses might record more or less information utilization in different cases.

2.8 Base Case Analysis[23]

The previous section proposed using an underspecifying lazy use analysis for making termination decisions during the analysis phase in order to minimize the probability of divergence of a partial evaluator. The downside of underspecification is the resulting larger equivalence classes can cause symbolic execution to terminate prematurely. In

[23]A complete design of base case analysis exists; however, it has not been fully implemented.

particular, two iterations of a recursion that could be executed to completion during the analysis phase might appear to be equivalent. Premature termination of symbolic execution can cause a partial evaluator to produce a residual recursion, rather than simple straight line code to compute the result. In the extreme case, premature termination might yield a residual recursion rather than just the code for the return value.

The iota function in Figure 2.49 is an example of a program for which utilization of an underspecifying lazy use analysis can lead to premature termination. Partial evaluation of `(iota 5)` ought to yield a program that just builds and returns the list `(0 1 2 3 4)`. However, an underspecifying lazy use analysis characterizes all of the iterations of the loop as being equivalent so symbolic execution is terminated before the final answer is determined. The culprit, so to speak, is the delta reduction `(= i n)`. In every iteration but the last, this expression returns the value `#f` since the loop index, `i`, is not equal to `5`. The actual information utilized is that the value of `i` is not equal to the value of `n`. The information domains in Figure 2.22 on page 31, repeated as Figure 2.50, cannot precisely capture the concept of inequality, so an approximation must be used. The overspecifying approximation is that the values of both `i` and `n` are used. Termination based on overspecification yields the desired result for partial evaluation of `(iota 5)`, but leads to divergence whenever `iota` is applied to an unknown value.[24] An underspecifying analysis states that only the integer property of `i` and `n` is used. This causes all of the applications of `=` to appear to be equivalent, regardless of the value of `i`, as long as `i` is not equal to `n`.

Base case analysis strives to mitigate, if not solve, the problem of premature termination. Base case analysis is grounded in the observation that no runtime recursion can terminate until a base case is reached. By extension symbolic execution of a recursion should not be terminated until at least one flow of control through the recursion has reached a base case. Otherwise, symbolic execution would in some sense be incomplete since at least one base case will have to be executed at runtime but no base case has been investigated during partial evaluation.

What precisely is a base case? All flows of control initiated by an application

[24]This is precisely the motivation for using an underspecifying analysis for termination decisions.

```
(define iota
  (lambda (n)
    (define loop
      (lambda (i)
        (if (= i n)
            '()
            (cons
             i
             (loop (1+ i))))))
    (loop 1)))
```

Figure 2.49: Iota function

Int	$=$	$\mathtt{0} + \pm\mathtt{1} + \pm\mathtt{2} + \cdots$	integers
		$\perp_{Int}$	unspecified integer
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
		$\perp_{Bool}$	unspecified boolean
Sym	$=$	$'\mathtt{a} + '\mathtt{b} + \cdots$	symbols
		$\perp_{Sym}$	unspecified symbol
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\perp_{Pair} \;\equiv \perp_{PEval} \times \perp_{PEval}$	unspecified pair
$Closure$	$=$	$Lambda \times Env$	closure values
		$\perp_{Clos}$	unspecified closure
Env	$=$	$(Id \rightarrow PEval)^{\star}$	environments
$Kval$	$=$	$Int + Bool + Nil + Pair + Closure$	known values
$Bots$	$=$	$\perp_{Int} + \perp_{Bool} + \perp_{Pair} + \perp_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \perp_{PEval}$	partial evaluation values
		$\perp_{PEval}$	unspecified value

Figure 2.50: Value domains for partial evaluation of a pure subset of Scheme (repeat of Figure 2.22)

fall in one of two classes. Some of the flows of control are terminated at recursive applications found to be equivalent to the initial application being considered. These flows of control are not base cases for the initial application being considered. All other flows of control returning a value for the initial application are considered to be bases cases. Even flows of control containing other terminated recursions based on different functions or applications are considered to be base cases for applications not terminated at recursive calls. This definition holds both for self and mutual recursions.

Base case analysis is a heuristic added on top of an underspecifying use analysis. Any time all of the control flow paths of a recursion are terminated because lazy use analysis decides the recursive applications are equivalent to earlier applications, premature termination has taken place.[25] To find a base case, symbolic execution of each terminated recursive applications is reinitiated. If the additional symbolic execution does not produce a base case, symbolic execution of the terminated recursive applications is once again reinitiated. This process of forcing analysis of additional iterations continues until at least one control flow path returns a value, as opposed to being terminated at a recursive function application.

The addition of base case analysis to lazy use analysis yields a partial evaluator that correctly unfolds `(iota 5)`. However, base case analysis is not a panacea. It is still possible for a partial evaluator to terminate recursions prematurely when a program contains multiple different base cases, only some of which have been investigated. Base case analysis cannot solve the multiple base case problem since it is undecidable whether all, or what subset of, base cases ought to be investigated for any set of argument values. The only absolute is that at least one base case must be reached if a recursion terminates. The drawback of base case analysis is that partial evaluation of some divergent recursions can fortuitously terminate in the absence of base case analysis, yet diverge once base case analysis is introduced. Whether this is of concern depends on what termination properties one desires.

[25]This assumes programs are not intentionally nonterminating.

2.9 Conclusion

Termination of partial evaluation requires limiting symbolic execution to investigation of a finite number of potential specializations. This can be achieved by dividing the potentially infinite set of possible applications of each function into a finite collection of subsets and allowing at most a finite number of potential specializations to be investigated for each subset. The selection of the subsets, or equivalence classes, determines how effective a partial evaluator is in producing quality residual code and whether it terminates.

Use analysis offers one approach to selection of equivalence classes. A simple eager use analysis was found to produce too many and too small equivalence classes to be an effective basis for termination. In response, lazy use analysis was created. The larger equivalence classes of the lazy analysis reduce the frequency of divergence. However, the decrease frequency of divergence comes at the cost of premature termination in some cases. In particular, some loops that could be completely executed or completely unrolled during partial evaluation are not fully analyzed. In response, base case analysis was developed to induce a partial evaluator to perform more symbolic execution than would result from lazy use analysis alone. The addition of base case analysis to lazy use analysis improves residual code.

Chapter 3

A Low Level Look at the Analysis Phase

This chapter investigates the low level details of the implementation of an analysis phase utilizing lazy use analysis for termination. These details are not necessary for a conceptual understanding of use analysis and may be skipped by the reader who desires a basic understanding, but does not require knowledge of the subtleties needed to implement the analysis. However, some of the details are necessary for a complete understanding of some of the sources of large resource consumption presented in Chapter 5.

This presentation is divided into two sections: preprocessing of a source program and actual analysis. During preprocessing, an input program is rewritten and converted into an intermediate form suitable as input to an analysis engine. During the analysis, symbolic execution is utilized to generate the information needed by a code generation phase in order to make the decisions necessary to produce high quality residual code.

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (-1+ n))))))
```

Figure 3.1: Factorial program (repeated from Figure 2.17)

3.1 Preprocessor

My preprocessor is composed of three passes: a front end, continuation passing style (CPS) conversion, and alpha conversion. The front end translates Scheme code, in the form of s-expressions, into a simple tree structured intermediate language based on the small number of primitive forms needed to represent Scheme. A standard CPS conversion is then performed on the output of the front end. Finally, the alpha conversion pass restructures the way variables are named, defined, and referenced.

A separate discussion of each of the three passes in the preprocessor appears in Sections 3.1.1- 3.1.3. The simple factorial program in Figure 2.17 on page 26, repeated as Figure 3.1, is used as an ongoing example throughout. The output of each of the stages of the preprocessor appears in Appendix A.

3.1.1 Front End

A simple pattern matching rewrite system suffices to parse Scheme. Due to the simplicity of this process, I do not present any rewrite rules or code for a pattern matching rewrite system.[1] I just show the tree structured intermediate language used for the output of my front end, discuss the two interesting rewrites, those for `letrec` and conditionals, and present the effects of feeding a factorial function through the front end.

The intermediate language used as output from the front end appears in Figure 3.2 as a Scheme definition. `Define-structure-collection` is a macro that builds a

[1]The interested reader can get more information about how to implement a rewriting system for Scheme in [1].

```
(define-structure-collection fe-code
  (fe-constant                      ;Self evaluating constants: booleans,
   value)                           ;characters, numbers, and strings
  (fe-reference                     ;Variable reference
   variable)                        ;Name of the referenced variable
  (fe-quote                         ;Quoted values
   expression)                      ;The quoted expression
  (fe-lambda                        ;Function creator
   name                             ;A name for identification purposes
   formals                          ;The names of the args
   body)                            ;The body of the function: a piece of
                                    ;fe-code
  (fe-definition                    ;A top level definition
   name                             ;The defined name
   value)                           ;The assigned value
  (fe-conditional                   ;A conditional
   predicate
   consequent
   alternative)
  (fe-application                   ;A function application
   function                         ;The function to apply
   args))                           ;A list of arguments
```

Figure 3.2: Front End Code

```
(letrec
  ((<variable 1> <init 1>) . . . )
  body)

≡

(let
  ((<variable 1> <unspecified>) . . . )
  (set! <variable 1> <init 1>)
  . . .
  body)
```

Figure 3.3: `Letrec` rewrite to `let` and `set!` form

group of structures.[2] The collection is assigned the name that is the first argument to `define-structure-collection`, in the case of Figure 3.2, `fe-code`. The subsequent arguments are lists whose heads are structure names and whose other elements are the field names of the structures. For example, the structure for lambda expressions is `fe-lambda`. It is composed of three fields: the name of the lambda, which is really just a comment, the formal parameters of the lambda, and the body of the lambda, which is another `fe-code` structure.

Those familiar with Scheme will note there are several primitive types missing in the `fe-code` definition. Assignment (`set!`) and a form to capture continuations (`call-with-current-continuation`) are not present because this work only addresses the functional subset of Scheme. However, the absence of a special form for creating recursive functions (`letrec`) is somewhat more interesting.

`Letrec` has two common implementations: *let and set* and the applicative order Y combinator[34, 20]. Let and set is the more frequently utilized technique for performance reasons. The let and set implementation as shown in Figure 3.3 is based on replacing `letrec`s with `let` bindings of the same variables to uninitialized or unspecified values and then binding each of those variables to the appropriate value using a `set!`.

The first problem with utilizing the let and set implementation of `letrec` for use

[2] My implementation produces constant time dispatchers for collections. This enables the creation of the equivalent of a case statement that executes in constant time, as opposed to time proportional to the number of cases weighted by their relative probabilities.

analysis is its dependence on the `set!` operator that does not exist in a functional subset of Scheme. However, the real problem is somewhat deeper and more interesting. An earlier version of this work tried to implement `letrec` using the let and set technique by introducing a `letrec-set!` operator solely for the purpose of implementing `letrec`. The tough question was what use dependences should be created for a `letrec-set!`.

Conceptually, `letrec`s create functions closed in the environment in which they are defined. This circularity makes the `letrec` form useful and is the source of the problem it poses. In use analysis the argument set of each function must be extended to include all variables a function is closed over that are referenced in the function's body. In the case of recursive function definitions introduced by `letrec`, this means recursive functions essentially become arguments of themselves. The problem is use of the function argument in a recursive application implies use of information about an aspect of the function itself. This circularity, if naively modeled with a circularity in the use dependence graph, can lead to divergence of the analysis phase due to a never ending propagation of use information along the circular path in the use dependence graph. Ideally, a fixed point of the use propagating around the circular portion of the use dependence graph needs to be computed. However, simple application of the technology used for the rest of use analysis would create use annotations that are a flattened versions of the circular use structure. As the use algorithm would propagate use information around a cycle repeatedly, the flattened representation would grow without limit, leading to divergence of the analysis.

Removing the circularity from use dependence graphs appears to be the simplest solution to the divergence problem. For simple recursions this might be achieved by removing recursive functions as virtual arguments of themselves. More complicated rewriting would be necessary for mutual recursions. Unfortunately, this naive approach does not lead to "correct" equivalences in all cases so it is not worth pursuing. To date, no correct set of use dependences has been identified for the let and set implementation of `letrec`.

The applicative order Y combinator implementation of `letrec` produces correct results in the absence of any special implementation techniques. This is because the applicative order Y combinator implementation is based on a rewriting of `letrec` that only depends on operators in the functional subset of Scheme shown in Figure 3.4.[3] There are two noteworthy aspects of the implementation of `letrec` in terms of the Y combinator. The first is that use analysis can handle the Y combinator at all. The Y combinator makes extensive use of higher-order functions. Many published analyses for making termination decisions in a partial evaluator have difficulty handling complex uses of higher-order functions. That use analysis of the Y combinator yields termination decisions powerful enough that `letrec`s can be replaced with uses of the Y combinator is therefore significant.

The second important point is that a partial evaluator based on use analysis removes all of the overhead of the Y combinator implementation of `letrec` before producing residual code in almost all cases. Basically, all of the added function applications get executed during the analysis phase and all that is left for runtime execution are the simple recursions apparent in the input program. The significant exception is when it is not possible during the analysis phase to determine what function will be applied by a given application at runtime due to a program using a closure created by `letrec` is a first-class manner. Depending on the code generation strategy used by a partial evaluator in such cases, it might be desirable to retain some auxiliary information about the rewriting of `letrec`s in order to produce residual code that does not contain any added overhead due to rewriting based on the Y combinator.

The bottom line is rewriting `letrec`s using the Y combinator yields a use analysis that produces good results. The execution costs of the increased computational complexity resulting from the rewriting are nearly all paid during the analysis phase when symbolic execution of the added function applications is performed. Little or no cost is incurred by the residual program. Discovery of a means of achieving an equivalent use analysis through the addition of appropriate arcs to a use dependence graph might improve the execution speed of partial evaluation, but is unlikely to yield significantly better residual code. Furthermore, since the let and set technique includes a side effect, it is possible that a use analysis capable of handling a language including arbitrary side effects would be necessary to handle this technique.

The `fe-definition` form in Figure 3.2 on page 80 is utilized only for top level

[3]For an intuitive explanation of the applicative order Y combinator, see [20].

```
(letrec
  ((<variable 1>
    (lambda (<x 11> . . .)
     <body 1>[<variable 1>, <variable 2>, . . .])
   (<variable 2>
    (lambda (<x 21> . . .)
     <body 2>[<variable 1>, <variable 2>, . . .]))
   . . .)
  <body>[<variable 1>, <variable 2>, . . .]))

≡

(let ((Y
       (lambda (f1 f2 . . .)
         (let ((h1
                (lambda (h1 h2 . . .)
                  (lambda (<x 11> . . .)
                    ((f1 (h1 h1 h2 . . .)
                         (h2 h1 h2 . . .)
                         . . .)
                     <x 11> . . .))))
               (h2
                (lambda (h1 h2 . . .)
                  (lambda (<x 21> . . .)
                    ((f2 (h1 h1 h2 . . .)
                         (h2 h1 h2 . . .)
                         . . .)
                      <x 21> . . .))))
               . . .)
           (list (h1 h1 h2 . . .)
                 (h2 h1 h2 . . .)))))
      (F1
       (lambda (f1 f2 . . .)
         (lambda (<x 11> . . .)
           <body 1>[f1 f2 . . .])))
      (F2
       (lambda (f1 f2 . . .)
         (lambda (<x 21> . . .)
           <body 2>[f1 f2 . . .]))))
  (let ((<funcs> (Y F1 F2 . . .)))
    <body>[(car <funcs>), (cadr <funcs>), . . .]))
```

Figure 3.4: `Letrec` rewrite using the applicative order Y combinator

definitions. Internal definitions, an optional Scheme syntax for introducing local function definitions in a lexical environment, are converted to their `letrec` equivalent as described in [10] and then rewritten using the Y combinator. However, mutual recursions amongst groups of functions introduced by top level definitions must be handled by introduction of a `letrec` either by hand by the programmer or automatically by the front end.

Processing of the factorial program in Figure 3.1 on page 79 by the front end yields the intermediate form shown in the first section of Appendix A. Although the output of the front end is a tree structured data structure, it has been represented as an S-expression in the appendix for ease of readability.

Note that `not` has been applied to the predicate of the conditional and the consequent and the alternative have been reversed as a result. The preprocessor introduces an application of `not` to all predicates because Scheme allows any expression as the predicate of a conditional, even if the expression does not return a boolean. The introduction of an application of `not` canonicalizes the results of all predicates to be booleans so `if` and other conditional special forms only have to deal with the details of implementing pure conditional operators expecting boolean predicates.

3.1.2 CPS Conversion

The CPS conversion pass uses a fairly standard implementation of CPS conversion, but with a couple of simple additions. Three special forms have been added to the standard intermediate language used to represent CPS converted Scheme programs, `clambda`, `throw`, and `exit-conditional`, all of which are explained below. A tree structured intermediate language, including these additions, used to represent programs resulting from the CPS conversion pass appears in Figure 3.5.

Most CPS converters use lambda expressions of a single argument for continuations and function applications to throw to continuations. For the purposes of partial evaluation, it is useful to maintain information regarding entry to and exit from functions. A program written in, or converted to, continuation passing style does not enter and return from functions, but instead just repeatedly enters one function after another until the program produces a final answer. In order to maintain information

```
(define-structure-collection cps-code
  (cps-constant                      ;Self evaluating constants: booleans,
   value)                            ;characters, numbers, and strings
  (cps-reference                     ;Variable reference
   variable)                         ;Name of the referenced variable
  (cps-quote                         ;Quoted values
   expression)                       ;The quoted expression
  (cps-lambda                        ;Function creator
   name                              ;A name for identification purposes
   formals                           ;The names of the args (including the
                                     ;continuation)
   body)                             ;The body of the function: a piece of
                                     ;cps-code
  (cps-definition                    ;A top level definition
   name                              ;The defined name
   value)                            ;The assigned value
  (cps-conditional                   ;A conditional
   predicate
   consequent
   alternative)
  (cps-exit-conditional              ;A throw to return a value from a
                                     ;conditional
   continuation                      ;The continuation to which to throw
   value)                            ;The return value
  (cps-application                   ;A function application
   function                          ;The function
   args)                             ;A list of arguments (including the
                                     ;continuation)
  (cps-clambda                       ;Continuation creator
   name                              ;A name for identification purposes
   formal                            ;The name of the arg
   body)                             ;The body of the continuation: a piece
                                     ;of cps-code
  (cps-throw                         ;A throw to a continuation
   continuation                      ;The continuation to which to throw
   value))                           ;The value supplied to the continuation
```

Figure 3.5: CPS Code

about entry to and exit from functions in an original source program, my implementation of CPS conversion introduces two new primitive forms `clambda` and `throw`. `Clambda` is a special version of `lambda` producing closures known to be continuations. `Throw` is a special form used to apply a continuation to a value for the purpose of exiting a function and returning the value to which the `clambda` is applied.

As presented in Section 2.6.1, a code dependence needs to be created between each value returned by a function and the function executed to create that value. Creating this form of use dependence requires being able to identify both the function that was applied and its return value. The applied function naturally appears on the stack of pending applications maintained by most partial evaluators; however, identifying when values are returned requires a little more effort. First, preprocessed input programs and the symbolic execution engine must be structured so that tail recursion elimination [2, 1] is not performed. Otherwise, information about function exits can disappear before use dependences for the return values is created. Retention of information about function returns in a program that has been CPS converted can be achieved by making each function exit explicit by introducing a throw to a continuation for every function exit. What would have been tail recursive exits from deeply nested sets of applications become a series of throws to continuations, each of which causes a return one level up the function application chain.

Second, the correspondences between function applications and return values must be identified in order to build the use dependences. The functions appear on the stack maintained by most partial evaluators, so all that is needed is a hook in the symbolic evaluation engine that detects when values are returned. The `throw` form supplies this hook. `Throw` identifies those throws to continuations used to return values from functions. The code generated to prevent tail recursion elimination is a series of `clambda`s; the body of each of which is a `throw` to another continuation. As a final note, if `clambda` were replaced by `lambda` and `throw` were replaced by function application in any program resulting from the CPS conversion pass of my preprocessor, the resulting code would be precisely what one would expect to see as the result of CPS conversion for an implementation not supporting tail recursion elimination.

Another use dependence that integrates dynamic control flow with data flow is the one between the value returned by a conditional and the value of the predicate of the conditional. Once a program has been CPS converted, the values of predicates are passed directly to the conditional special form. However, there is no identification of return values from conditionals. My CPS converter introduces the `exit-conditional` special form for that purpose. `Exit-conditional` operates just like `throw`. It throws to a continuation that is its first argument, supplying it with the value that is its second argument. When symbolic execution encounters an `exit-conditional` form, it is handed the return value of the conditional as its second argument. To build a use dependence, the symbolic execution engine needs the value of the predicate. My symbolic execution engine maintains a stack of pending conditionals as part of the stack of pending function applications. When an `exit-conditional` is performed, information about the predicate of the conditional is available in the conditional frame at the top of the stack. `Exit-conditional` creates a use dependence and pops the conditional frame off the stack just like `throw` creates a use dependence and then pops an application frame off of the stack.

Once the factorial example has been fed through the CPS conversion pass, it looks as is shown in the second section of Appendix A. Note that CPS conversion of `fact` introduces a `lambda` expression that surrounds the conditional. CPS conversion introduces the lambda to avoid duplicating the continuation of the conditional. Effectively, it uses a `let` to bind the continuation to a name and then refers to the continuation by that name in both the consequent and the alternative of the conditional. Of course, a `let` is just the application of a `lambda` to a series of values, so the output of the CPS conversion pass contains an application and a `lambda`. Furthermore, since all returns from applications are explicit in the output of the CPS conversion pass, `throw`s must be introduced in the code for both the consequent and the alternative for the return from the introduced `lambda`.

3.1.3 Alpha Conversion

Calling the third pass of the preprocessor alpha conversion is a misnomer. The alpha conversion pass does not perform alpha conversion, instead it changes the ways variables are defined, named, and referenced. The most significant change during the alpha conversion pass is in the representation of lambdas. Because use analysis requires the argument set of closures to include all of the lexically inherited bindings in the body of a closure, the alpha conversion phase modifies the representation of lambdas to include two sets of arguments. The formal parameters are the names of the variables that store the arguments to which a closure is applied. The inheriteds are the variables the lambda was closed over that are referenceable from the body of the lambda. A new definition for `lambda` appears in the alpha code definitions in Figure 3.6.

Once lambdas have been augmented to include the inherited bindings, true lexical environments are no longer needed. Each closure carries with it a flattened version of the lexical hierarchy. The arguments to which a closure is applied become local variables in the environment in which a closure's body is evaluated, and all of the lexically referenceable variables become inherited lookups in the same environment. All variable references in the body of a lambda are performed by looking up the variable binding in the single environment created for execution of the closure's body.

Even though there can be no ambiguity regarding to what variable a given reference refers, for pedagogical reasons I chose to retain the distinction between local and inherited variables. I also chose to create two different types of variable references, one for local bindings for arguments and the other to inherited bindings from values in the virtual lexically enclosing environments. As a result, the `reference` special form in Figure 3.5 has been replaced by `local-reference` and `inherited-reference` forms in Figure 3.6.

The `local-reference` and `inherited-reference` forms differ from their `reference` predecessor in their inclusion of two new fields: `offset` and `single-reference?`. `Offset` is a numeric offset into the vector of bindings in an environment. It is the second half of the lexical level/offset representation used in many compilers [2]. The lexical level portion is unneeded since the lexical hierarchy has been eliminated. Looking up a reference requires knowing only whether the reference is to a local or an inherited binding and its offset.

The `single-reference?` field indicates whether a reference is the only one to a given binding that appears anywhere in the body of a function. When a binding

```
(define-structure-collection alpha-code
  (alpha-constant             ;Self evaluating constants: booleans,
   value)                     ;characters, numbers, and strings
  (alpha-local-reference      ;Variable reference to a local -
                              ;defined in this scope
   variable                   ;Name of the referenced variable
   offset                     ;Offset into the list of locals
   single-reference?)         ;#t, if unique reference
  (alpha-inherited-reference  ;Variable reference to an inherited - Always
                              ;inherited from the directly enclosing scope
   variable                   ;Name of the referenced variable
   offset                     ;Offset into the list of inheriteds
   single-reference?)         ;#t, if unique reference
  (alpha-quote                ;Quoted values
   expression)                ;The quoted expression
  (alpha-lambda               ;Function creator
   name                       ;A name for identification purposes
   formals                    ;The names of the args (including the continuation)
   inheriteds                 ;References to inherited values from enclosing
                              ;scopes
   body)                      ;The body of the function: a piece of alpha-code
  (alpha-definition           ;A top level definition
   name                       ;The defined name
   global-number              ;The offset into the list of globals
   value)                     ;The assigned value
  (alpha-conditional          ;A conditional
   predicate
   consequent
   alternative)
  (alpha-exit-conditional     ;A throw to return a value from a conditional
   continuation               ;The continuation to which to throw
   value)                     ;The return value
  (alpha-application          ;A function application
   function                   ;The function
   args)                      ;A list of arguments (including the continuation)
  (alpha-clambda              ;Continuation creator
   name                       ;A name for identification purposes
   formal                     ;The name of the arg
   inheriteds                 ;References to inherited values from enclosing
                              ;scopes
   body)                      ;The body of the continuation: alpha-code
  (alpha-throw                ;A throw to a continuation
   continuation               ;The continuation to which to throw
   value)))                   ;The value supplied to the continuation
```

Figure 3.6: Alpha Code

has at most one reference, use analysis can perform optimizations that reduce memory consumption and increase speed of the analysis. This field is recording during preprocessing for utilization later in the analysis phase.

Other changes between CPS code and alpha code are all direct extensions of the ones presented so far. `Clambda`s have an added field for inheriteds that is akin to the one for `lambda`s. `Definitions` have an added field called `global-number` that is an offset into the vector of bindings in the global environment. The other code representations remain unchanged.

At the conclusion of the alpha conversion pass, the factorial program generated looks as shown in third section of Appendix A.

3.1.4 Conclusion

The preprocessor performs a fairly straightforward rewriting of source programs into a tree structured intermediate language suitable for use by the rest of the analysis phase. The steps in the conversion process are a simple rewriting of s-expressions into a standard set of Scheme forms by the front end, a replacement of `letrec` by uses of the Y combinator, CPS conversion, and finally the removal of lexical structure and variable renaming through the introduction of what I have termed *lexical arguments.* The next section describes the uses of symbolic execution to perform the real work of the analysis phase.

3.2 Symbolic Execution and Computation of Lazy Use

This section presents details on the implementation of the analysis phase of a partial evaluator whose termination mechanism is based on lazy use analysis. The discussion is separated into three major portions. The first describes the operation of symbolic execution and the formation of a use dependence graph. The second presents how uses are propagated along the arcs in a use dependence graph, how this can result in the detection that two values that were previously believed to be equivalent are actually not equivalent, and how this realization can lead to further symbolic execution. Finally, a description of how base case analysis integrates with the rest of lazy use analysis is presented.

3.2.1 Symbolic Execution

This explanation begins with a presentation of the data structures utilized during symbolic execution and their purposes. Next an explanation of how the various special forms in Scheme are handled during symbolic execution is given. This is followed by a deeper discussion of the most interesting form, function application. Of particular note is how the handling of primitive procedures differs from that of *compound procedures*[4] [24] and how a generalized return value is generated when symbolic execution of a recursion is terminated based on an equivalence detected by use analysis.

The Data Structures Utilized During Symbolic Execution

The final version of a source program after rewriting by the preprocessor is loaded by the symbolic execution engine for processing by the remainder of the analysis phase. The tree structured intermediate language used to represent programs once they are loaded for symbolic execution appears in Figure 3.7. There are two significant differences between the code representation used in performing analysis of a program and the final version of the code as produced by the preprocessor. The first is that the `lambda` and `clambda` forms both have an added field used in tracking all of the closures and continuations created by execution of those forms. The code generation phase uses the information to identify all the different potential specializations of a `lambda` or `clambda` investigated during the analysis phase. The second difference is the addition of the `primitive` special form. In the representation of code used in performing the actual analysis, the `primitive` form is used for primitive procedures and the `lambda` form is used only for compound procedures.

In addition to code, the other key objects are symbolic values, use dependences, and use annotations. A brief discussion of the representations selected for each follows. Symbolic values are represented as shown in Figure 3.8. The first field stores a characterization of a runtime value and the second field a characterization of the lazy use annotation for that value.

Figures 3.10 and 3.11 show the representations for values that are manipulated by

[4]Those procedures defined in terms of other primitive and compound procedures

```
(define-structure-collection pe-code
  (pe-constant                ;Self evaluating constants: booleans,
   value)                     ;characters, numbers, and strings
  (pe-local-reference         ;Variable reference to a local
   variable                   ;Name of the referenced variable
   offset                     ;Offset into the list of locals
   single-reference?)         ;#t, if unique reference
  (pe-inherited-reference     ;Variable reference to an inherited
   variable                   ;Name of the referenced variable
   offset                     ;Offset into the list of inheriteds
   single-reference?)         ;#t, if unique reference
  (pe-quote expression)       ;Quoted values
  (pe-lambda                  ;Function creator
   name                       ;A name for identification purposes
   formals                    ;The names of the args (including the continuation)
   inheriteds                 ;References to the values inherited from enclosing
                              ;scopes
   body                       ;The body of the function: a piece of pe-code
   closures)                  ;Closures formed by evaluating this lambda
  (pe-definition              ;A top level definition
   name                       ;The defined name
   global-number              ;The offset into the list of globals
   value)                     ;The assigned value
  (pe-conditional             ;A conditional
   predicate
   consequent
   alternative)
  (pe-exit-conditional        ;A throw to return a value from a conditional
   continuation               ;The continuation to which to throw
   value)                     ;The return value
  (pe-application             ;A function application
   function                   ;The function
   args)                      ;A list of arguments (including the continuation)
  (pe-clambda                 ;Continuation creator
   name                       ;A name for identification purposes
   formal                     ;The name of the arg
   inheriteds                 ;References to the values inherited from enclosing
                              ;scopes
   body                       ;The body of the continuation: a piece of pe-code
   continuations)             ;Continuations formed by evaluating this clambda
  (pe-throw                   ;A throw to a continuation
   continuation               ;The continuation to which to throw
   value)                     ;The value supplied to the continuation
  (pe-primitive               ;The body of a primitive
   function))                 ;A function of 2 args (stack and env)
                              ;executed to produce the return value
```

Figure 3.7: Partial Evaluation Code

```
(define-structure
  symbolic-value
  value                              ;The value in the heap
  use-profile)                       ;The use description for this object
```

Figure 3.8: Representation for Symbolic Values

Int	$=$	$\mathtt{0} + \pm\mathtt{1} + \pm\mathtt{2} + \cdots$	integers
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
Sym	$=$	$'\mathtt{a} + '\mathtt{b} + \cdots$	symbols
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$Sval \times Sval$	pairs
$Closure$	$=$	$Lambda \times Env$	closure values
Env	$=$	$(Id \rightarrow Sval)^{\star}$	environments
$Sval$	$=$	$Int + Bool + Nil + Pair + Closure$	scheme values

Figure 3.9: Value domains for a pure subset of Scheme (repeat of Figure 2.21)

the symbolic execution engine. These representations correspond to the value domains shown in Figure 2.21 on page 31, repeated in Figure 3.9, with one minor addition: the new `continuation` value type has been introduced to represent continuations as a distinct type of object from the other closures.

The other significant data structures represent use annotations and use dependences. The `use-profile` field of each symbolic value points to a `use-profile` of the form shown in Figure 3.12. The `use-annotation` field represents the use annotation for the corresponding symbolic value as determined by lazy use analysis. `Dependences` contains a list of the other use profiles that depend on this one. These are the arcs in a use dependence graph. When the memoized annotation of the `use-profile` changes through modification of the `use-annotation` field, a message is sent to every use profile in the dependences list informing the use profiles at the other end of the arcs of the use change. Since the amount of use recorded for each node can only monotonically increase, memoization plus update messages sent when an annotation changes is guaranteed to produce correct annotations, and to do so efficiently.

Use annotations are represented by the data structures defined in Figure 3.13. Not surprisingly, these data structures are virtually identical to the ones for representing

```
(define-structure-collection pe-values
  (pe-boolean                       ;Booleans - #t or #f
   value)                           ;#t or #f
  (pe-bottom-boolean)               ;An unspecified boolean
  (pe-symbol                        ;Symbols
   value)                           ;The symbol's value (e.g., 'a)
  (pe-bottom-symbol)                ;An unspecified symbol
  (pe-integer                       ;Integers
   value)                           ;The integer's value (e.g., 23)
  (pe-bottom-integer)               ;An unspecified integer
  (pe-closure                       ;Closures
   parent-env                       ;The closing environment
   lambda                           ;The lambda expression executed to
                                    ;create this closure
   applications)                    ;A list of structures describing
                                    ;applications of this closure
  (pe-bottom-closure)               ;An unspecified closure
  (pe-continuation                  ;Continuations
   clambda                          ;The clambda expression executed to
                                    ;create this continuation
   stack                            ;The stack to be installed when a throw
                                    ;to this continuation is performed
   parent-env                       ;The closing environment
   throws)                          ;A list of structures describing
                                    ;throws to this closure
  (pe-bottom-continuation)          ;An unspecified continuation
  (pe-pair                          ;A cons cell
   car                              ;A symbolic value representing the car
   cdr)                             ;A symbolic value representing the cdr
  (pe-null)                         ;The empty list
  (pe-bottom-value)                 ;An unspecified value
  (pe-top-value                     ;A special value that can only result
                                    ;from generalization of no values
   specializations))                ;A list of specialization points,
                                    ;application descriptions, for which
                                    ;this bottom is a generalized return
                                    ;value
```

Figure 3.10: Representations for Values during Symbolic Execution

```
(define-structure
  pe-env
  creator                           ;Either a closure, a continuation, or
                                    ;'GENERALIZATION
  parent                            ;The parent environment
  locals                            ;A vector of symbolic values
  inheriteds                        ;A vector of symbolic values
  comparison-cache)                 ;A cache of environments to which this
                                    ;one is currently being compared
```

Figure 3.11: Representation for Environments during Symbolic Execution

```
(define-structure
  use-profile
  use-annotation                    ;A use structure defining the type of
                                    ;use
  dependences)                      ;A list of links reflecting how changes
                                    ;in use should be propagated.
```

Figure 3.12: A Data Structure for Use Information

values as shown in Figure 3.10. The last important set of data structures are those used to represent arcs in the use dependence graph as shown in Figure 3.14. For clarity, each of the different types of use dependences is briefly discussed.

An `id-dependence` signifies that whatever use is made of the value that is the source of the dependence, the identical use is made of the value that is the sink of the dependence. In other words, this is the identity use dependence. The `id-dependence` is utilized when an argument is passed to a function. A new symbolic value is created for the actual parameter in the execution environment in which the function's body is evaluated. The new symbolic value has an `id-dependence` to the argument symbolic value to which the function was applied. The motivation for copying the symbolic value and creating an `id-dependence` is that while every use of the value in the body of the applied function implies corresponding use of the argument to which the function was applied, the converse is not necessarily true. If two functions are applied to the same symbolic value, the use of the argument value is the union of the uses of that value by the two functions. However, the uses each function makes

```
(define-structure-collection use-types
  (use-boolean                          ;Booleans - #t or #f
   value)                               ;#t or #f
  (use-bottom-boolean)                  ;An unspecified boolean
  (use-symbol                           ;Symbols
   value)                               ;The symbol's value (e.g., 'a)
  (use-bottom-symbol)                   ;An unspecified symbol
  (use-integer                          ;Integers
   value)                               ;The integer's value (e.g., 23)
  (use-bottom-integer)                  ;An unspecified integer
  (use-closure                          ;Closures
   closure)
  (use-bottom-closure)                  ;An unspecified closure
  (use-continuation                     ;Continuations
   continuation)
  (use-bottom-continuation)             ;An unspecified continuation
  (use-pair                             ;A cons cell whose identity is used
   cons)
  (use-bottom-pair)                     ;An unspecified pair
  (use-null)                            ;The empty list
  (use-bottom-value)                    ;An unspecified, but needed value
  (unused))                             ;A completely unused value (bottom)
```

Figure 3.13: Data Structures for Use Annotations

```
(define-structure-collection use-dependence-types
  (id-dependence              ;An id dependence between two use profiles
   source
   sink)
  (cons-dependence            ;A dependence between a cons cell and
   source                     ;its car and cdr objects
   car-sink
   cdr-sink)
  (car-dependence             ;A dependence between the car of a cons
   source                     ;cell and the cons cell
   sink)
  (cdr-dependence             ;A dependence between the cdr of a cons
   source                     ;cell and the cons cell
   sink)
  (closure-dependence         ;A dependence between a closure and
   source                     ;the values inherited from its env
   sinks)                     ;A vector of symbolic values
  (application-dependence     ;A dependence between an inherited
   closure                    ;value and the associated closure
   source
   sink
   sink-offset)
  (continuation-dependence    ;A dependence between a continuation and the
   source                     ;values inherited from its environment
   sinks)
  (throw-dependence           ;A dependence between an inherited value
   continuation               ;and the associated continuation
   source
   sink
   sink-offset)
  (generic-dependence         ;A generic dependence between any two use profiles
   source
   sink
   sink-asserter)
  (resumption-dependence      ;A dependence to resume execution if
   source                     ;the use of an object changes
   resumption-lock
   resumption-thunk)          ;The thunk to execute to resume exec
  (specialization-dependence  ;An id dependence between an argument of a
   source                     ;specialization and an argument of a recursive call
   sink-use-profile           ;to that specialization.  Used to complete the use
   sink-pe-value              ;cycle for the finite representation of a loop.
   thread))
```

Figure 3.14: Data Structures for Use Dependences

of its formal parameters need to be retained separately as well. Those uses must not be conflated between different functions, so a new use profile, and therefore a new symbolic value, is created for each parameter when a function application is performed during symbolic execution.[5]

The potential to conflate information in the absence of copying of symbolic values is demonstrated by the code in Figure 3.15. If the symbolic value for the cons cell to which `independent-uses-cons-cell` is bound were not copied before being passed to `use-car` and `use-cdr` then `independent-uses-cons-cell`, `use-car-cons-cell`, and `use-cdr-cons-cell` would all be bound to the same symbolic value. Assuming `use-car` makes use of some information about the car of the cons cell and `use-cdr` makes use of some information about the cdr of the cons cell, the single symbolic value would contain the union of those uses. This is correct for the symbolic value to which `independent-uses-cons-cell` is bound, but not for either `use-car-cons-cell` or `use-cdr-cons-cell`. Imagine a termination decision is being made for a recursive call to `use-car`. It should only be effected by uses of `use-car-cons-cell` taking place as a result of execution of the body of `use-car`; however, if the symbolic value for `independent-uses-cons-cell` is not copied prior to passing it as an argument then the termination decision is effected by the irrelevant uses of the cdr of the cons cell in `use-cdr`. Copying of symbolic values supplied as arguments insures uses from different callees are aggregated in the caller, but uses by the caller or other callees are not conflated with those of individual callees.

A `cons-dependence` is created by the `cons` primitive function. It represents the use dependence between the symbolic value for a cons cell returned by an application of the `cons` function and the symbolic values for the car and cdr arguments to which `cons` was applied. A `cons-dependence` signifies that any use made of the car of a cons cell is also made of the symbolic value of the first argument to `cons`. Similarly, any use of the cdr of a cons cell is also made of the symbolic value of the second argument to `cons`.

[5]This rule doesn't hold when there can only be a single reference to a symbolic value during symbolic execution. In this case, use of a newly created symbolic value is guaranteed to produce identical results to use of the value of which it is a clone; and, the copying of the symbolic value can be avoided. The `single-reference?` field in the reference code types in Figure 3.7 on page 94 supports this performance optimization.

```
(define independent-uses
  (lambda (independent-uses-cons-cell)
    . . .
    (use-car independent-uses-cons-cell)
    . . .
    (use-cdr independent-uses-cons-cell)
    . . .
    ))

(define use-car
  (lambda (use-car-cons-cell)
    . . .
    (car use-car-cons-cell)
    . . .
    ))

(define use-cdr
  (lambda (use-cdr-cons-cell)
    . . .
    (cdr use-cdr-cons-cell)
    . . .
    ))
```

Figure 3.15: Why symbolic values need to be copied when being passed as arguments

```
(define curried-addition
  (lambda (a)
    (lambda (b)
      (+ a b)))))
```

Figure 3.16: A curried version of the addition function

The `car-dependence` and `cdr-dependence` are in some sense the inverses of the `cons-dependence`. They are generated by the `car` and `cdr` primitive functions. They represent that use of a symbolic value returned by an application of `car` or `cdr` implies the same use of the corresponding field of the symbolic value for the cons cell to which `car` or `cdr` was applied.

A `closure-dependence` represents the relationship between the symbolic value for a closure and the symbolic values in the environment in which a lambda expression was evaluated in order to form the closure. When a use is made of an inherited value in the body of a function, a `closure-dependence` ensures the same use is recorded for the corresponding value in the environment in which the lambda expression was closed. A simple example of a `closure-dependence` arises from the curried addition function in Figure 3.16. Loading of the definition of `curried-addition` causes the function definition to be evaluated and the outermost lambda expression to be converted into a closure. The closure is closed in an environment that has a binding for `+`. A `closure-dependence` is created at the time `curried-addition` is closed between `curried-addition` and the symbolic value for `+` in the enclosing environment. Similarly, when `curried-addition` is applied to some value, another closure is produced. This closure is closed in an environment that has a local definition for `a` and an inherited definition for `+`. The `closure-dependence` created when the inner lambda is closed represents the relationship between the symbolic value for the closure of the inner lambda and the symbolic values for `a` and `+` in the environment in which the inner lambda was closed.

An `application-dependence` codifies the relationship between symbolic values for inherited values in the execution environment for the body of a closure and the corresponding symbolic values in the closure object. In this case the `source` of the dependence is a symbolic value in the environment created for execution of the body of the applied closure and the `sink` is the closure that was applied. The `sink-offset` field specifies which of the symbolic values in the environment portion of the closure is dependent on the symbolic value in the current execution environment for the closure body.

Returning to the `curried-addition` example in Figure 3.16, when the closure for `curried-addition` is applied to a value, a new environment is created for the execution of the closure body. That environment contains a local binding for `a` and an inherited binding for `+`. An `application-dependence` is created linking the symbolic value for `+` in the new environment to the symbolic value for the `curried-addition` closure. The combination of the `application-dependence` and the previously described `closure-dependence` causes any use made of `+` in the body of `curried-addition` to be propagated to the corresponding symbolic value for `+` in the lexically enclosing environment. Similarly, when the closure resulting from applying `curried-addition` to a first value is applied to a second value, a new environment is built that contains a local binding for `b` and inherited bindings for `+` and `a`. `Application-dependence`s are created for both of the inherited values at the time of application. The eventual use of the value of `+` in the body of the closure of the inner lambda propagates over two `application-dependence`s, two `closure-dependence`'s, and through two different closures in getting between the point of use of the closure and its point of definition.

The `continuation-dependence` and `throw-dependence` serve identical purposes for continuations and throws as the `closure-dependence` and `application-dependence` serve for closures and applications. This is not surprising due to the symmetries between `lambda` and `clambda`, closures and continuations, and applications and throws.

A `generic-dependence` is utilized to represent all of the various other forms of dependences between symbolic values. Most of these arise from the execution of primitive functions. For example, the function `not` can be applied to any type of value. If the argument is not a boolean, then the result of applying `not` is determined by the type of the argument. If the argument is a boolean, then the result is determined by the boolean value of the argument. In the first case, use of the boolean value of the result implies use of the type of the argument. A `generic-dependence` can be utilized to represent this relationship between the boolean value of one symbolic value and the type of another symbolic value.

The `Sink-asserter` field is a function that encodes how the use of information about `source` implies use of other information about `sink`. When the use annotation of the source changes, the desired effect on the use annotation of the sink is determined by applying the `sink-asserter` function to the new use annotation for the source.

The `resumption-dependence` field contains a daemon that notifies the symbolic execution engine when a recursion that has been previously terminated might need to be reinitiated. `Resumption-dependence`s detect changes in the use annotation of one of two symbolic values previously deemed equivalent. A change in an annotation indicates the values might not be equivalent.

The `specialization-dependence` handles a subtle case of use analysis. The analysis needs to generate uses of the arguments of final recursive function applications that are not performed when symbolic execution of a recursion is terminated. A generalized return value is created for the application, and symbolic execution continues from the point of the function return; but, the terminated application is not performed so symbolic execution is not performed utilizing the supplied arguments.

Since the application not performed is presumed to be equivalent to prior applications initiating the iterations of the terminated recursion, the logical solution is to utilize the uses made of the arguments of earlier applications and assert the same uses about the arguments to the application not performed. Asserting some type of use of the arguments to the function application never performed is necessary to ensure appropriate use equivalence of symbolic values elsewhere in a program. In the absence of asserting some use of these arguments, appropriate use cannot be propagated to the symbolic values utilized to generate the arguments. Consequently, values that ought to be use equivalent would have different use annotations. The bottom line is that `specialization-dependence`s are needed between arguments to applications not performed due to termination and the corresponding arguments of the applications initiating earlier iterations of the recursion to which the final application is deemed equivalent.[6]

[6]It has never been proven that using `specialization-dependence`s yields the desired equivalences in all cases. In practice this approach seems to work well on all programs investigated, while the absence of `specialization-dependence`s yields poor results. A better way of generating use

Symbolic Execution of the Scheme Forms

The evaluation framework utilized for symbolic execution in my partial evaluator is based on a virtual machine with four pieces of state: an expression, a stack, an environment, and a thread. The expression is the piece of code (`pe-code`) currently being executed. The stack maintains a stack of pending function calls and conditionals utilized by the analysis in deciding when to terminate recursions. The environment is a flat environment model as shown in Figure 3.11 on page 97. It is flat in the sense that all of the inherited values have been copied into each environment so no references up the lexical hierarchy are needed to get values defined in enclosing scopes. Finally, there is a thread. Threads are used to keep track of the different possible flows of control resulting from execution of undecidable control flow operators. For example, a conditional for which the predicate's value cannot be determined during symbolic execution yields two alternative flows of control. Distinct threads are created for each possible flow of control, and the threads are analyzed separately. For the remainder of this discussion, the explanations of how symbolic execution handles different forms only consider expressions, stacks, and environments, suppressing the details of threads. Thread handling adds considerable complexity to the code, but is not terribly enlightening.

Symbolic execution of a `pe-constant` expression creates a new symbolic value, with the `value` field being the appropriate form of `pe-value` for the constant expression executed and the `use-profile` being initialized with a use annotation of `unused`. For a constant, the stack and environment are irrelevant.

Symbolic execution of `pe-quote` is basically identical to that of `pe-constant`. `pe-quote` creates an unused symbolic value with the appropriate quoted type of `pe-value` as its value.

Pseudocode for performing symbolic execution of a `pe-local-reference` appears in Figure 3.17. First, the symbolic value to which the `pe-local-reference` refers is looked up in the environment, `env`. If the reference is the only possible reference to the specified symbolic value, then the symbolic value returned by the lookup in

annotations for the arguments of terminated applications may exist.

```
(define sym-exec-local-reference
  (lambda (ref stack env thread)
    (let ((referenced-value
           (pe-env-local-ref
            env
            (pe-local-reference.offset
             ref))))
      (if (pe-local-reference.single-reference? ;If unique reference:
           ref)
          referenced-value                       ;Return the reference symbolic
                                                 ;value
          (make-id-symbolic-value                ;Else, return a copy of the
                                                 ;value
          referenced-value)))))
```

Figure 3.17: Pseudocode for performing symbolic execution of a `pe-local-reference`

the environment can be returned by the `pe-local-reference`.[7] In the more general case, the result of the lookup must be copied. A new symbolic value is created having the identical value to the symbolic value returned by the lookup. Its use annotation is initialized to be `unused`. Most importantly, an `id-dependence` is created between the new symbolic value and the one looked up in the environment. This dependence is stored in the use profile of the newly created symbolic value. It ensures that use asserted about the newly created symbolic value is propagated along the use dependence graph, and it is also asserted about the symbolic value of which the new one is a copy.

Symbolic execution of a `pe-lambda` forms a symbolic value for the resulting closure as shown in the pseudocode in Figure 3.18. It then adds a closure dependence from the newly formed closure to the inherited values from the enclosing environment. The symbolic values for the inherited values are identified by performing a value lookup in the enclosing environment for each of the inherited parameters in the `pe-lambda` description.

Symbolic execution of the `pe-definition` form is straightforward as is shown by the pseudocode in Figure 3.19. The symbolic value to be assigned is determined by

[7]The optimization of only copying symbolic values when necessary was explained in a previous section.

```
(define sym-exec-lambda
  (lambda (a-lambda stack env thread)
    (let* ((symbolic-value                ;Build symbolic value
            (make-unused-pe-object
             (make-pe-closure
              env
              a-lambda)))
           (closure-use-profile           ;Extract use profile
            (symbolic-value.use-profile
             symbolic-value)))
      (add-use-dependence                 ;Add closure dependence
       closure-use-profile
       (make-closure-dependence
        closure-use-profile
        (list->vector
         (pe-lambda-inheriteds-map
          (lambda (ref)
            (symbolic-value.use-profile
             (pe-env-ref
              env
              ref)))
          (pe-lambda.inheriteds
           a-lambda)))))
      symbolic-value)))
```

Figure 3.18: Pseudocode for performing symbolic execution of a `pe-lambda`

```
(define sym-exec-definition
  (lambda (a-define stack env thread)
    (pe-global-env-set!
     (pe-definition.global-number       ;The global to be installed
      a-define)
     (sym-exec-expr                     ;The value to install
      (pe-definition.value
       a-define)
       stack
       env
       thread))))
```

Figure 3.19: Pseudocode for performing symbolic execution of a `pe-definition`

performing symbolic execution of the body of the definition. The name to which this symbolic value is to be bound has already been converted by the preprocessor into an offset into the table of global definitions, so the symbolic value is just assigned to the appropriate slot in the data structure for global definitions.

Symbolic execution of a `pe-conditional` is the first interesting case. Pseudocode for processing conditionals appears in Figure 3.20. The first step is to generate a symbolic value for the predicate of the conditional through symbolic execution. If the boolean value of the predicate can be determined during the analysis phase, symbolic execution can proceed with either the consequent or the alternative, as appropriate. However, first a conditional frame must be placed on the stack. The frame is utilized when the consequent or alternative returns a value. A use dependence will be generated between the return value and the symbolic value for the predicate found in the conditional frame, as described in Section 2.6.1.

When the boolean value of the predicate is not decidable during the analysis phase, both the consequent and the alternative must be investigated. Again, a conditional frame is first placed on the stack. Next, the information needed for analyzing the consequent is placed on a queue of threads to be processed through symbolic execution. Finally, symbolic execution of the alternative is performed. A new thread is created for symbolic execution of both the consequent and the alternative so the analysis phase can keep track of the separate contributions of the different flows of control.

```
(define sym-exec-conditional
  (lambda (conditional stack env thread)
    (let* ((predicate-symbolic-value        ;Symbolically execute the predicate
            (sym-exec-expr
             (pe-conditional.predicate
              conditional)
             stack
             env
             thread))
           (predicate-value                 ;Extract the value field
            (pe-object.value predicate)))
      (if (known-runtime-truth-value?       ;If runtime truth value predicate
           predicate-value)                 ;is known at analysis time:
          (begin
            (push-cond-frame
             stack
             (make-conditional-desc
              predicate-symbolic-value
              #t))                          ;The conditional was decidable
            (sym-exec-expr                  ;Symbolically execute either the
             (if (runtime-truth-value       ;consequent or the alternative
                  predicate-value)
                 (pe-conditional.consequent
                  conditional)
                 (pe-conditional.alternative
                  conditional))
             stack
             env
             thread))
          (begin
            (push-cond-frame                ;Frame for this conditional
             stack
             (make-conditional-desc
              predicate-symbolic-value
              #f))                          ;The conditional was undecidable
            (scheduler-queue-enqueue        ;Schedule alternative for later
             (pe-conditional.alternative    ;execution
              conditional)
              stack
              env
              (make-child-thread            ;Separate thread needed for
               thread))                     ;alternative
            (sym-exec-expr                  ;Evaluate the consequent immediately
             (pe-conditional.consequent
              conditional)
             stack
             env
             (make-child-thread             ;Separate thread needed for
              thread)))))))                 ;consequent
```

Figure 3.20: Pseudocode for performing symbolic execution of a `pe-conditional`

Pseudocode for symbolic execution of a `pe-exit-conditional` appears in Figure 3.21. The first step is getting information about the conditional being exited from off the stack. Next, the continuation of the conditional and the value to be returned by the conditional must be computed through symbolic execution. If the boolean value of the predicate of the conditional was decidable during symbolic execution, a use dependence must be created between the return value of the conditional and the value of the predicate. This dependence relates use of information about the result of the conditional to use of the boolean value of the predicate utilized in making a control flow decision to execute either the consequent or the alternative. Symbolic execution of a `pe-exit-conditional` form concludes by throwing to the continuation of the conditional, passing the continuation the return value of the conditional.

The process of throwing to a continuation is shown in the pseudocode in Figure 3.22. After looking up both the continuation to which to throw and the clambda from which the continuation was formed, the real work begins. A new environment in which the body of the continuation will be evaluated is created. This environment consists of a new binding for the value on which the continuation is invoked plus those bindings that are inherited from the environment in which the continuation was formed. Once the new environment is generated, use dependences are created between the inherited values in the new environment and the continuation object from which they were generated. Finally, the actual execution of the continuation takes place by recursively invoking the symbolic execution engine. The code to be executed is the body of the continuation to which the throw is being performed. By definition, the stack to be used after a throw is the one captured at the time the continuation to which the throw is performed was created; and, the environment in which the body of the continuation should be evaluated is the newly created environment.

The most interesting case for symbolic execution is function application. Pseudocode for processing a `pe-application` appears in Figure 3.23. First, the closure to be applied is generated through symbolic execution and its different components are extracted. Then, the arguments to which the closure will be applied are created through symbolic execution. An environment in which the body of the closure can be executed is created based on the argument values and the values inherited from the closing environment of the closure.

```
(define sym-exec-exit-conditional
  (lambda (exit-cond stack env thread)
    (let* ((cond-desc                         ;Get the conditional off the stack
            (pop-cond-frame
             stack))
           (cont-symbolic-value               ;Compute the continuation
            (sym-exec-expr
             (pe-exit-conditional.continuation
              exit-cond)
             stack
             env
             thread))
           (returned-symbolic-value           ;Compute the return value
            (sym-exec-expr
             (pe-exit-conditional.value
              exit-cond)
             stack
             env
             thread)))
      (if (conditional-desc.decidable?        ;If the conditional being exited
           cond-desc)                         ;wasdecidable:
          (add-any->bool-dependence           ;Add a dependence between the
           (symbolic-value.use-profile        ;return value of the conditional
            returned-symbolic-value)          ;and the boolean value of the
           (symbolic-value.use-profile        ;predicate
            (conditional-desc.predicate
             cond-desc))
           (pe-boolean.value
            (symbolic-value.value
             (conditional-desc.predicate
             cond-desc)))))
      (perform-invoke-continuation
       cont-symbolic-value
       returned-symbolic-value
       stack
       env
       thread))))
```

Figure 3.21: Pseudocode for performing symbolic execution of a `pe-exit-conditional`

```
(define perform-invoke-continuation
  (lambda (cont-symbolic-value arg-symbolic-value stack env thread)
    (let* ((cont-value                      ;Get the value portion of the
            (symbolic-value.value           ;continuation
             cont-symbolic-value))
           (cont-clambda                    ;Get the clambda from which the
            (pe-continuation.clambda        ;continuation was formed
             cont-value))
           (new-env                         ;A new environment built from the
            (build-child-environment
             (pe-continuation.parent-env    ;Parent environment
              cont-value)
             (vector arg-symbolic-value)    ;Local values
             (pe-clambda.inheriteds         ;Inherited values
              cont-clambda))))
      (for-each-inherited-value             ;Create a use dependence for every
       (lambda (inherited-symbolic-value    ;inherited value
                offset)
         (add-use-dependence
          (symbolic-value.use-profile       ;Store the dependence with the
           inherited-symbolic-value)        ;inherited value
          (make-throw-dependence            ;Dependence type
           cont-symbolic-value
           (symbolic-value.use-profile      ;Source
            inherited-symbolic-value)
           (symbolic-value.use-profile      ;Sink symbolic value
            cont-symbolic-value)
           offset)))                        ;Field within the sink
       (pe-clambda.inheriteds
        cont-clambda))
      (sym-exec-expr                        ;Symbolically execute
       (pe-clambda.body                     ;The continuation body
        cont-clambda)
       (pe-continuation.stack               ;Using the stack captured by
        cont-value)                         ;the continuation
       new-env                              ;And the newly created environment
       thread))))
```

Figure 3.22: Pseudocode for throwing to a continuation

```
(define sym-exec-application
  (lambda (application stack env thread)
    (let* ((function-symbolic-value     ;Get the closure being applied
             (sym-exec-expr
              (pe-application.function
               application)
              stack
              env
              thread))
            (function-value             ;And its associated value field
             (symbolic-value.value
              function-symbolic-value))
            (function-lambda            ;Get the lambda from which the closure
             (pe-closure.lambda         ;being applied was formed
             function-value))
            (arg-symbolic-values        ;Symbolically evaluate each of the args
             (map (lambda (arg)
                    (sym-exec-expr
                     arg
                     stack
                     env
                     thread))
                  (pe-application.args
                   application)))
            (new-env                    ;Create a new environment for the
             (build-child-environment   ;application from the
              (pe-application.parent-env ;Parent environment
               function-value)
              (list->vector             ;Local values
               arg-symbolic-values)
              (pe-lambda.inheriteds     ;Inherited values
               function-lambda))))
      (let ((terminate?
              (terminate-execution?     ;If symbolic execution of a recursion
               function-value           ;has reached a fixed-point
               stack
               new-env)))
        (if terminate?
            (process-termination
             terminate?
             new-env
             function-symbolic-value
             arg-symbolic-values)
            (process-application
             function-symbolic-value
             arg-symbolic-values
             stack
             new-env))))))
```

Figure 3.23: Pseudocode for performing symbolic execution of a application

```
(define terminate-execution?
  (lambda (function stack env)
    (let* ((compliant-applications
            (stack-appl-search
             (lambda (appl-stack-frame)
               (use-compliant-application?
                function
                env
                appl-stack-frame))
             stack))
      (application-search
       use-equivalent-applications?
       compliant-applications)))))
```

Figure 3.24: Pseudocode for checking whether symbolic execution of a recursion should be terminated

The critical step in symbolic execution of a function application is deciding whether to perform the application. The `terminate-execution?` function implements the termination mechanism of use analysis. It decides whether an application is distinct from all previous applications and should therefore by analyzed through symbolic execution, or whether symbolic execution of a recursion should be terminated. If the application should be performed, then symbolic execution proceeds with the `process-application` function. Otherwise, termination is handled by `process-termination`. The one subtlety is that the `terminate-execution?` function returns the two equivalent previous applications, when a decision is made to terminate. The equivalent applications are then supplied to the `process-termination` function.

The details of `terminate-execution?` appear in the pseudocode in Figure 3.24. First, the stack is searched for all previous applications of equivalent functions compliant with the current application. That is, all applications of closures of the same lambda to arguments with use annotations compliant with the argument values supplied in the current application. The compliant applications are then searched to find any two having equivalent use annotations for their arguments. Assuming equivalent applications are located on the stack, the two most recent equivalent applications are returned. If no pair of equivalent applications is found, then `#f` is returned.

The `application-search` function is a special version of `for-each` applying the closure supplied as its first argument to pairs of elements of the list that is its second argument until the closure argument returns a true value for some pair. Once a true value is found, it is returned as the result of `application-search`. `Use-equivalent-applications?` takes two application descriptions as arguments and compares the environments generated for execution of those applications in order to determine whether corresponding bindings in the two environments have equivalent use annotations.

The `process-application` function as shown in Figure 3.25 implements symbolic execution of a function application.[8] Of note is the creation of an `application-dependence` between the symbolic value for each inherited value in the environment created for performing the function application and the closure object being applied. Just prior to initiating symbolic execution of the body of the closure, an application frame for the current application is pushed onto the stack.

The final case in symbolic execution of applications is when a recursion is terminated. In this case, symbolic execution proceeds from the point at which a value would be returned by the function application not analyzed due to termination. Pseudocode for performing termination is shown in Figure 3.26. The first step is probably the most important. Detection of two recursive applications deemed equivalent based on their having equivalent use annotations for their corresponding arguments caused a termination decision to be made. Should the use annotations of any of the symbolic values involved in those comparisons change in the future, the equivalence decision might be invalidated, necessitating the resumption of symbolic execution of the recursion beginning with the terminated application. Monitoring of changes in the use annotations of the critical symbolic values is implemented by the `annotate-to-resume-exec-on-use-change` function. This function creates a daemon that detects changes in the use annotations of values using `resumption-dependence`s.

Before symbolic execution can proceed from the point at which a value would have been returned by a terminated application, a generalized return value is needed

[8]This pseudocode does not address the details of the case in which the function to be applied is not uniquely known during partial evaluation.

```
(define process-application
  (lambda (function-symbolic-value arg-symbolic-values stack new-env)
    (let* ((function-value                ;Get the closure value
            (symbolic-value.value
             function-symbolic-value))
           (function-lambda               ;Get the lambda from which the
            (pe-closure.lambda            ;closure being applied was formed
            function-value))
      (for-each-inherited-value           ;Create a use dependence for every
       (lambda (inherited-symbolic-value  ;inherited value
                offset)
         (add-use-dependence
          (symbolic-value.use-profile     ;Store the dependence with the
           inherited-symbolic-value)      ;inherited value
          (make-application-dependence    ;Application type
           function-symbolic-value
           (symbolic-value.use-profile    ;Source
            inherited-symbolic-value)
           (symbolic-value.use-profile    ;Sink symbolic value
            function-symbolic-value)
           offset)))                      ;Field within the sink
       (pe-lambda.inheriteds
        function-clambda))
      (push-appl-frame                    ;Push a frame for this application
       stack                              ;onto the stack
       (make-application-desc
        function-value
        new-env
        #f))                              ;Not starting a specialization
                                          ;due to termination
      (sym-exec-expr                      ;Symbolically execute the closure
       (pe-lambda.body                    ;body
        function-lambda)
       stack
       new-env
       thread)))))
```

Figure 3.25: Pseudocode for performing symbolic execution of an application

```
(define process-termination
  (lambda (eqv-appl-descs new-env function-symbolic-value arg-symbolic-values)
    (annotate-to-resume-exec-on-use-change    ;Enable daemons
     eqv-appl-descs                           ;that will cause termination
     function-symbolic-value                  ;to be rechecked on use changes
     stack
     new-env
     thread)
    (if (start-new-specialization?            ;If a specialized version of the
         eqv-appl-descs                       ;the function still must be
         function-symbolic-value              ;investigated to create the
         new-env)                             ;generalized return value:
        (process-build-generalization
         eqv-appl-descs
         new-env
         function-symbolic-value
         arg-symbolic-values)
        (let ((generalized-return-value       ;Else, if a specialization has been
               (generalized-return-value      ;completed
                eqv-appl-descs)))
          (add-changed-generalization-daemon
           eqv-appl-descs
           new-env
           stack)
          (add-any->code-dependence           ;Set up the dependence of
           (symbolic-value.use-profile        ;function on the generalized
            generalized-return-value)         ;return value
           (symbolic-value.use-profile
            function-symbolic-value))
          (link-uses-of-start&end-of-recursion
           eqv-appl-descs
           new-env)
          (perform-invoke-continuation        ;Actually evaluate the continuation
           (continuation-arg
            arg-symbolic-values)
           generalized-return-value
           stack
           new-env
           thread)))))
```

Figure 3.26: Pseudocode for terminating a recursion

for the application. If the generalized return value has not already been computed, then a new specialization must be analyzed in order to generate the value. The details of the process of generating a generalized return value as implemented by `process-build-generalization` are discussed in a future section.

Once a generalized return value is available, all that remains is to implement the function return. A generic use dependence is created so any uses of the return value of the terminated application imply use of the code portion of the closure to have been applied in the terminated application. As described in Section 2.6.1, use dependences are also created linking the symbolic values of the arguments of the terminated application to those of the applications initiating previous iterations of the terminated recursion. Finally, a function return is performed by throwing to the continuation of the terminated application using the generalized return value.

The one final detail is the call to `add-changed-generalization-daemon`. Its purpose is related to the creation of generalized return values and will be discussed in a future section.

Symbolic execution of `pe-clambda` is virtually identical to that of `pe-lambda` and needs no further discussion. However, symbolic execution of `pe-throw` as implemented by the pseudocode in Figure 3.27 is critical. `Sym-exec-throw` is a wrapper that computes the continuation and argument objects using symbolic execution and then calls `perform-sym-exec-throw` to do the real work. `Perform-sym-exec-throw` first checks whether the continuation represents a return of the result of a program to the read/eval/print loop. The details of returning a value to the read/eval/print loop are not particularly interesting and have been omitted. In the more common case, a generic use dependence is created to represent that any use of the return value of a closure implies the code of the closure producing the result has been utilized. Next, the value returned is recorded. All that remains is to invoke the continuation using the `perform-invoke-continuation` function previously presented in Figure 3.22 on page 112.

The final form is the `pe-primitive` form used for implementing primitives. Pseudocode for this form as shown in Figure 3.28 is trivial because all of the interesting aspects are implemented as part of the primitives, themselves.

```
(define sym-exec-throw
  (lambda (throw stack env thread)
    (let ((cont-symbolic-value           ;Get the continuation object
           (sym-exec-expr
            (pe-throw.continuation throw)
            stack
            env
            thread))
          (arg-symbolic-value            ;Symbolically evaluate the arg
           (sym-exec-expr
            (pe-throw.value throw)
            stack
            env
            thread)))
      (perform-sym-exec-throw
       cont-symbolic-value
       arg-symbolic-value
       stack
       env
       thread))))

(define perform-sym-exec-throw
  (lambda (cont-symbolic-value arg-symbolic-value stack env thread)
    (if (not (repl-continuation?         ;If this is not a throw back to the
              cont-symbolic-value ))     ;repl:
        (begin
          (add-any->code-dependence      ;Set up the dependence between the
           (symbolic-value.use-profile   ;Return value and
            arg-object)
           (symbolic-value.use-profile   ;The code of the closure that is
            (application-desc.closure    ;returning the value
             (top-appl-desc stack))))
          (add-return-value              ;Add the return value to the list of
           (top-appl-desc stack)         ;values returned by this application
           arg-symbolic-value
           thread)))
          (perform-invoke-continuation
           cont-symbolic-value
           arg-symbolic-value
           stack
           env
           thread))
        (schedule-next-thread))))        ;Else, start execution of another thread
```

Figure 3.27: Pseudocode for symbolic execution of a `pe-throw`

```
(define sym-exec-primitive
  (lambda (prim stack env thread)
    ((pe-primitive.function
      prim)
     stack
     env
     thread)))
```

Figure 3.28: Pseudocode for symbolic execution of a `pe-primitive`

Symbolic Execution of Primitive Procedures

Symbolic execution of the `pe-application` form handles analysis of compound procedures as shown in the previous section. The real work of a program gets done in the primitives. Since none of the details of the implementation of primitives is apparent in the pseudocode for symbolic execution of the `pe-primitive` form, one must examine the implementations of primitives to appreciate what is taking place.

The `make-builtin-primitive` function for which pseudocode appears in Figure 3.29 is used to create a primitive function to be loaded into the global environment. The important aspect of the code in Figure 3.29 is the creation of a `pe-lambda` for every primitive. Symbolic execution of the `pe-lambda` builds the environment for executing the body of the primitive. The body of the primitive as implemented by `sym-exec-func` is able to assume the environment has already been built correctly and only needs to handle the details of implementing a particular primitive.

Pseudocode for an implementation of the `integer?` primitive appears in Figure 3.30. The body of the implementation is wrapped in a call to `perform-sym-exec-throw`, as defined in Figure 3.27, causing the value returned by the primitive to be passed to the continuation of the application of `integer?` via a throw. The continuation to be invoked is the second argument to `integer?` and is extracted from the environment created for execution of the body of `integer?` by the `arg-2` function. The main portion of the implementation of `integer?` is devoted to computing the symbolic value representing the result and building the use dependences for the result value. This process proceeds as follows.

```
(define make-primitive
  (lambda (name num-args sym-exec-func)
    (make-pe-definition
     (builtin-global-number
      name)
     (make-pe-lambda
      (make-lambda-args
       num-args)
      (make-lambda-inheriteds          ;A primitive has no inherited args
       '())
      (make-pe-primitive
       sym-exec-func)))))
```

Figure 3.29: Pseudocode for symbolic execution of a `make-builtin-primitive`

First, the argument to `integer?` is extracted from the environment. If the argument is a bottom value, then it is not possible to determine what boolean result should be returned by `integer?` until runtime; therefore, a symbolic value whose value is an unspecified boolean ($\bot_{Bool}$) is created. In this case, the only use dependence necessary is one representing that any use of the unspecified boolean return value implies the actual return value must be computed at runtime. If any information is available about the argument to `integer?`, then the argument's type can be determined so the return value of `integer?` is computable during the analysis phase. Once the return value is computed, the appropriate use dependence is created. If the return value is `#t`, any computation making use of trueness of the return value also makes use of the integer type of the argument. This is reflected in the generic dependence built by `add-b->type-dependence`. If the return result is `#f`, the information used in computing the result is that the argument is not of type integer. Unfortunately, this fact cannot be represented by any use annotation in the domains in Figure 2.22 on page 31, repeated in Figure 3.31. To say the type of the argument was used in determining the return value would be an overspecification of use. The correct, conservative approximation for a lazy use analysis for termination is to state that any use of the return value implies only that the argument must be computed at runtime. That is, any use of the result implies bottom value use of the argument. This dependence must be generated regardless of the value returned by `integer?`. It is built and installed by `add-any->bottom-value-dependence`. Finally, `result-symbolic-value`

```
(define make-integer?
  (lambda ()
    (make-builtin-primitive
     'integer?
     1                                          ;1 arg
     (lambda (stack env thread)
       (perform-sym-exec-throw                  ;Throw to the
        (arg-2 env)                             ;Continuation with
        (let* ((arg-symbolic-value              ;The value
                (arg-1 env))
               (arg-value                       ;Get components of the arg
                (symbolic-value.value arg))
               (arg-use-profile
                (symbolic-value.use-profile arg)))
          (if (pe-bottom-value? arg-value)      ;If arg's value unknown:
              (let ((result-symbolic-value
                     (make-unused-symbolic-value  ;Return an unspecified boolean
                      (make-pe-bottom-boolean))))
                (add-any->bottom-value-dependence ;Add a control dependence
                 (symbolic-value.use-profile    ;between the result and the arg
                  result-symbolic-value)
                 arg-use-profile)
                result-symbolic-value))
              (let* ((result-value              ;Else, argument known so compute
                      (integer-type? arg-value))  ;result
                       arg-value))
                     (result-symbolic-value     ;Create a known boolean object
                      (make-unused-pe-object
                       (make-pe-boolean
                        result-value))))
                (if result-value                ;If the return value is #t:
                    (add-b->type-dependence     ;Add a dependence between the
                     (pe-object.use-profile     ;value of the returned boolean
                      result-symbolic-value)    ;and the  type of the arg
                     arg-use-profile
                     (pe-object.value
                      result-symbolic-value)))
                (add-any->bottom-value-dependence ;Otherwise, should add a
                 (pe-object.use-profile         ;b->bottom-value dependence
                  result-symbolic-value)        ;which is subsumed by this
                 arg-use-profile)               ;dependence
                result-symbolic-value))
     stack
     env
     thread)))))
```

Figure 3.30: Pseudocode for symbolic execution of the primitive `integer?`

Int	$=$	$\mathtt{0} + \pm\mathtt{1} + \pm\mathtt{2} + \cdots$	integers
		$\bot_{Int}$	unspecified integer
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
		$\bot_{Bool}$	unspecified boolean
Sym	$=$	$'\mathtt{a} + '\mathtt{b} + \cdots$	symbols
		$\bot_{Sym}$	unspecified symbol
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\bot_{Pair} \equiv \bot_{PEval} \times \bot_{PEval}$	unspecified pair
$Closure$	$=$	$Lambda \times Env$	closure values
		$\bot_{Clos}$	unspecified closure
Env	$=$	$(Id \rightarrow PEval)^{\star}$	environments
$Kval$	$=$	$Int + Bool + Nil + Pair + Closure$	known values
$Bots$	$=$	$\bot_{Int} + \bot_{Bool} + \bot_{Pair} + \bot_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \bot_{PEval}$	partial evaluation values
		$\bot_{PEval}$	unspecified value

Figure 3.31: Value domains for partial evaluation of a pure subset of Scheme (repeat of Figure 2.22)

is returned by `integer?` by passing it as an argument to `perform-sym-exec-throw`.

The `cons` function shown in Figure 3.32 serves as another interesting example of the implementation of a primitive. As for `integer?`, the body of the implementation of `cons` is a throw to the continuation of `cons` using the value returned by `cons`. The throw is implemented by an application of `perform-sym-exec-throw`. Its first argument is the continuation of the application to `cons`, which is the third argument to which `cons` was applied. The continuation is extracted from the environment built for executing `cons` by `arg-3`. The value returned by `cons` is the second argument supplied to `perform-sym-exec-throw`.

In order to compute the return value, first the two arguments to be concatenated by `cons` to produce the result are extracted using `arg-1` and `arg-2`. Next, the symbolic value for the pair to be returned is produced. Finally, use dependences are needed to relate the symbolic value for the result to the symbolic values for the arguments. A `pe-cons-dependence` is created by the `make-cons-dependence` function relating uses of either the car or cdr of the resulting pair to either the first or second arguments supplied to `cons`, respectively. Two generic dependences are created by `add-any->bottom-value-dependence` signify that if any use is made of the return value, the arguments utilized in creating the pair must be computed at runtime. The process of supplying the result to `perform-sym-exec-throw` is completed by returning the symbolic value for the return value from the `let` form used to produce it.

Generating Generalized Return Values

The pseudocode for building a generalized return value is shown in Figure 3.33. First, a generalized environment and function[9] are computed based on the three applications. The generalized environment contains the generalizations of the arguments and the generalizations of the inherited values from the environments of the closures. Next, the continuation of the final application not performed is inserted into the appropriate place in the generalized environment since the desire is for the generalized return value to be returned to that continuation, not a generalization of several continuations. A description of the generalized application is then placed on the stack. Finally, symbolic execution of the generalized application is performed.

The second half of the process of forming a generalized return value was already presented in the pseudocode for `process-termination` in Figure 3.26 on page 117 and for `perform-sym-exec-throw` in Figure 3.27 on page 119. The call to `add-changed-generalization-daemon` in `process-termination` builds a daemon that looks for changes in a generalized return value. If the generalized return value changes, the daemon causes symbolic execution of the associated computation to be repeated for the new generalized return value.

[9]The termination mechanism utilized by my system differs from those of many other systems in that two applications can be found to be equivalent even if the closures applied are not identical. This is because a closure is really just a piece of code and an environment. Consequently, the applications of two closures formed from the same lambda expression can be determined to be equivalent if the applications make equivalent uses of corresponding values in the two closures' environments. This fact is mentioned to explain why my system must compute a generalized closure to be used in performing a generalized application. A generalized closure is created by taking the greatest lower bound of several closures. The generalized closure consists of the common code and a generalization of corresponding fields in the closures' environments. As one might suspect, my system forms the generalized closure from the combination of three different closures: the two used in the equivalent applications and the one in the application about to be performed when the termination decision was made.

```
(define make-cons
  (lambda ()
    (make-builtin-primitive
     'cons
     2                                      ;2 args
     (lambda (stack env thread)
       (perform-sym-exec-throw              ;Throw to the
        (arg-3 env)                         ;Continuation with the value
        (let* ((arg1 (arg-1 env))           ;Get args
               (arg2 (arg-2 env))
               (result-symbolic-value
                (make-unused-symbolic-value
                 (make-pe-pair
                  arg1
                  arg2)))
               (result-use-profile
                (symbolic-value.use-profile
                 result-symbolic-value)))
          (add-use-dependence
           result-use-profile
           (make-cons-dependence
            result-use-profile
            (symbolic-value.use-profile
             arg1)
            (symbolic-value.use-profile
             arg2)))
          (add-any->bottom-value-dependence ;Add a control dependence between
           result-use-profile               ;the result and the args
           (symbolic-value.use-profile
            arg1))
          (add-any->bottom-value-dependence
           result-use-profile
           (symbolic-value.use-profile
            arg2))
          result-symbolic-value)
        stack
        env
        thread)))))
```

Figure 3.32: Pseudocode for symbolic execution of the primitive `cons`

```
(define process-build-generalization
  (lambda (eqv-appl-descs new-env function-symbolic-value arg-symbolic-values)
    (let ((generalized-env                ;Build generalized environment
           (make-generalized-environment
            eqv-appl-descs
            new-env))
          (generalized-function            ;Build generalized closure
           (make-generalized-function
            eqv-appl-descs
           function-symbolic-value)))
      (pe-env-local-set!                  ;Set the continuation to be the one
       generalized-env                    ;supplied to this call rather than
       (-1+                               ;a generalization of the
        (pe-env-num-locals                ;continuations
         generalized-env))
       continuation)
      (push-appl-frame                    ;Push a frame for this application
       stack                              ;onto the stack
       (make-application-desc
        generalized-function
        generalized-env
        eqv-appl-descs))                  ;The two applications for which
                                          ;the specialization is being built
      (sym-exec-expr                      ;Symbolically execute the closure
       (pe-lambda.body                    ;body - identical to that of the
        (pe-closure.lambda                ;generalized function
         (symbolic-value.value
          function-symbolic-value)))
        stack
        generalized-env
        thread))))
```

Figure 3.33: A function for that builds a generalized return value

```
(define add-return-value
  (lambda (appl-desc return-value thread)
    (add-return-value-to-appl-desc!          ;Store the new return value
     appl-desc
     return-value)
    (if (computing-generalized-return-value?
         appl-desc)
        (let* ((old-generalization           ;Get the old generalization
                (generalized-return-value
                 appl-desc))
               (new-generalization           ;Compute the new generalization
                (make-unused-symbolic-value  ;Store the result in an object
                 (symbolic-value.value       ;that has no use links to the
                  (generalize                ;original values to prevent use
                   old-generalization        ;from feeding through the
                   return-value)))))         ;generalized return value to the
                                             ;other return values
          (if (not (generalization-eqv?      ;If the generalization has changed
                    old-generalization
                    new-generalization))
              (begin
                (set-generalized-return-value! ;Store the new generalization
                 appl-desc                   ;This may activate changed
                 new-generalization)))))))   ;generalization daemons and cause
                                             ;symbolic execution of some
                                             ;computations to be repeated
```

Figure 3.34: A function for processing the values returned by applications

The application of `add-return-value` in `perform-sym-exec-throw` does more than just keep a list of values returned by functions as becomes apparent from the pseudocode in Figure 3.34. After storing each return value, it is determined whether the current return value requires changing the generalized return value as follows. A new generalized return value is computed by generalizing the existing generalized return value and the current return value. The existing generalized return value and the new generalized return value are then compared. If the two generalized return values are not equivalent, the new generalized return value is saved. The process of saving the new generalized return value may activate changed generalization daemons and cause symbolic execution of some computations to be repeated utilizing the new generalized return value.

3.2.2 Computation of Lazy Use via Use Propagation

The first part of the analysis phase consists of performing symbolic execution of an input program based on a partial specification of its inputs. During symbolic execution, use dependences are created between different symbolic values. However, no symbolic value is ever assigned a use annotation other than $\perp$ (completely unused). As a result, no use is propagated along any of the use dependences created.

The process of assigning other use annotations to values is initiated after symbolic execution has completed. Assignment of new use annotations is initiated by assigning $\perp_{PEval}$ use to every one of the values returned by a program[10]. Once use annotations have been assigned to the return value(s), use changes propagate along use dependences to change the annotations on other symbolic values.

As use annotations are modified, `resumption-dependence`s may be activated. `Resumption-dependence`s are designed to keep track of when uses of either of two symbolic values previously deemed to be use equivalent change. The activation of a `resumption-dependence` does not guarantee two symbolic values are not really use equivalent, it just indicates that it might be a possibility. It is possible the use annotations of both symbolic values are going to change to the same value. For example, when the iota function in Figure 2.49 on page 75, repeated in Figure 3.35, is applied to $\perp_{Int}$, the argument `i` for every application of `loop` is initially annotated as unused. After the return results have all been annotated with $\perp_{PEval}$ use, all of the arguments to which `loop` is applied will also eventually be annotated with $\perp_{PEval}$ use. However, each change in use of an argument to `loop` activates a `resumption-dependence`. Since the propagation of use along use dependences is a sequential process, there are points in time at which the use annotations will not be equivalent.

Because use annotations can become temporarily not equivalent, even though they may quiesce to equivalent values, symbolic execution is not resumed the instant a resumption dependence is activated by a change in a use annotation. Instead, information about all activated resumption dependences is retained until all use annotations have been updated based on the full set of use dependences and the use annotations

[10]Because symbolic execution can investigate multiple different possible flows of control through a program, symbolic execution may return a family of possible return values, as opposed to a single value.

```
(define iota
  (lambda (n)
    (define loop
      (lambda (i)
        (if (= i n)
            '()
            (cons
             i
             (loop (1+ i))))))
    (loop 1)))
```

Figure 3.35: Iota function (repeat of Figure 2.49)

asserted for the return values. At that point, each resumption dependence activated is checked to see if two symbolic values previously deemed to be use equivalent no longer have equivalent use annotations. In cases in which symbolic values are found no longer to be use equivalent, symbolic execution of any recursion terminated based on the perceived equivalence must be resumed from the point at which the recursion was terminated.

Resumption of symbolic execution of a terminated recursion consists of two distinct actions. The results of performing symbolic execution of the continuation of the terminated recursion using a generalized return value must be marked as defunct. In particular, information generated during the process of performing symbolic execution of the continuation should not be passed to the code generation phase.[11]

The second aspect of resuming analysis of a recursion is to ensure symbolic execution of the recursion is initiated starting at the point at which it was terminated. This is achieved by creating a thread to perform the desired application and placing it on the queue of flows of control requiring analysis. Once all of the resumption dependences have been considered and the appropriate threads placed on the queue of pending threads to be analyzed, the only remaining step is for the analysis phase to return control to the symbolic execution engine to proceed in the fashion outlined in Section 3.2.1.

Eventually, the symbolic execution engine once again will run out of threads to be

[11]The results of the analysis may be retained by the analysis phase in case it is later determined the identical computation needs to be analyzed for some other purpose.

evaluated. $\perp_{PEval}$ use of return values can once again be asserted. This time, both the new return values generated by resumed symbolic execution and the old return values for which $\perp_{PEval}$ use was previously asserted must be considered. The new return values are handled identically to the way return values were handled the first time use was asserted about return values. For old return values, it is critical that the use asserted about the old values be propagated over any new use dependences created during the second pass of symbolic execution. How this is achieved is an implementation detail. [12]

Once the second pass of asserting use about return values has been completed, there may or may not be more previously terminated recursions for which renewed symbolic execution is desired. If no recursions require renewed symbolic execution, the basic lazy use analysis algorithm has been completed and use annotations are available for all symbolic values. However, if more symbolic execution is required, the symbolic execution engine is reactivated and the process continues alternating between symbolic execution and use propagation until there is no longer any need indicated for further symbolic execution.

[12]In my implementation I record every use dependence created linking an unused symbolic value to a symbolic value that already has a use annotation other than unused. During the use propagation stage, my system considers every one of those dependences to be a source for a chain of possible use updates and processes those just as if some new use had been asserted about the sources symbolic value of those dependences.

3.2.3 Base Case Analysis[13]

Keeping track of whether any base cases have been investigated is directly supported by the infrastructure described in Section 3.2.1. The pseudocode for `perform-sym-exec-throw` in Figure 3.27 on page 119, repeated in Figure 3.36, includes an application of the function `add-return-value`. Since a call to `perform-sym-exec-throw` is made for every value returned by a function, `add-return-value` records every value returned for every flow of control resulting from every function application. By tracing the flow of control back from the return of a result to its point of creation, it is possible to determine whether a return value originated from a generalized return value generated for a terminated recursion (of the function application from which the value is being returned) or whether the value represents the result of a base case.

During base case analysis, all of the return values for each function application initiating a terminated recursion are checked to determine if any of them results from a base case of the source function. If none of the return values results from a base case, then further analysis of that recursion is desirable. This is achieved by creating a thread to renew execution of every terminated recursive application of the function for which no base case has been investigated. The renewal threads are placed on the queue of flows of control requiring analysis much as was done for the processing of `resumption-dependence`s as presented in Section 3.2.2.

If at the end of base case analysis, no threads to renew execution of any loops have been created, then the analysis phase is completed. If base case analysis identifies recursions to be analyzed further, symbolic execution is reinitiated. At the end of symbolic execution, use propagation is once again required. Once both symbolic execution and use propagation are no longer indicated, base case analysis is once again performed. This pattern of passes repeats until base case analysis no longer finds any recursions requiring further analysis. A flow chart for the complete analysis phase appears in Figure 3.37

[13]The analysis described in this section has been designed, but has not been implemented as an integral part of my partial evaluator.

```
(define perform-sym-exec-throw
  (lambda (cont-symbolic-value arg-symbolic-value stack env thread)
    (if (not (repl-continuation?          ;If this is not a throw back to the
              cont-symbolic-value ))      ;repl:
        (begin
          (add-any->code-dependence       ;Set up the dependence between the
           (symbolic-value.use-profile    ;Return value and
            arg-object)
           (symbolic-value.use-profile    ;The code of the closure that is
            (application-desc.closure     ;returning the value
             (top-appl-desc stack))))
          (add-return-value               ;Add the return value to the list of
           (top-appl-desc stack)          ;values returned by this application
           arg-symbolic-value
           thread)))
          (perform-invoke-continuation
           cont-symbolic-value
           arg-symbolic-value
           stack
           env
           thread))
        (schedule-next-thread))))         ;Else, start execution of another thread
```

Figure 3.36: Pseudocode for symbolic execution of a `pe-throw` (repeat of portion of Figure 3.27)

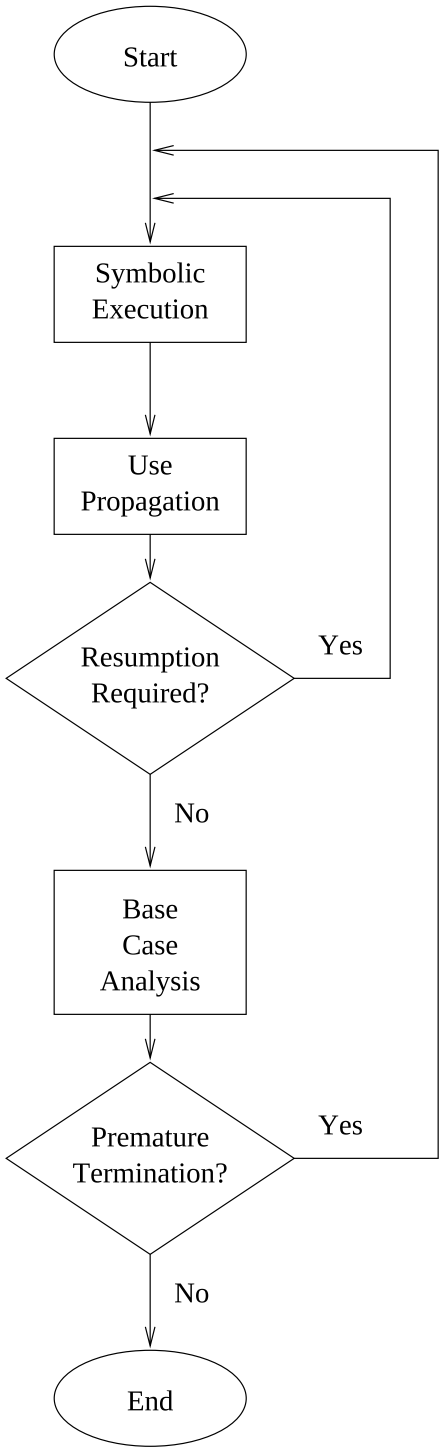

Figure 3.37: A flow chart for the analysis phase

Chapter 4

Related Work

This chapter discusses related work in the area of partial evaluation. The presentation is divided into three major sections. First, termination mechanisms for both online and offline partial evaluation are described, along with their advantages and disadvantages. This is followed by a discussion of the role of the continuation passing style (CPS) transformation in various partial evaluators including my own. Finally, the pros and cons of delayed commitment partial evaluation of the form utilized by Osgood and myself is presented.

4.1 Alternative Termination Mechanisms

Use analysis was developed in response to a small number of common forms of source code not all handled appropriately by any existing partial evaluator. All existing termination mechanisms either failed to guarantee convergence or failed to yield acceptable residual code for at least one of the "problem" cases. This section begins by presenting two simple programs demonstrating the termination problems motivating the development of use analysis. It proceeds to discuss different automatic termination mechanisms used by a variety of online and offline partial evaluators. The systems and termination mechanisms presented are Mix, Finiteness Analysis (and its extensions), Similix, and Fuse. Why each of the termination mechanisms fails to handle effectively at least one of the sample programs, along with the other advantages and disadvantages of each of the termination mechanisms, is explained. Tables comparing the different termination mechanisms appear in Figures 4.1 and 4.2. This discussion does not address termination based on user annotation (e.g., Schism [11]), termination of partial evaluation of other types of programming languages (e.g., logic programming [18]), or termination of supercompilation [42, 45].

4.1.1 Termination Problems and Examples

This section includes two examples, each of which presents a termination problem for some partial evaluators. The first example demonstrates a condition called *changing static values under dynamic control* that can lead to divergence. Attempts to prevent divergence in this case are complicated by the fact that premature termination and therefore poor quality residual code can result for other partial evaluations. The second example exemplifies a case in which many partial evaluators terminate prematurely yielding poor quality residual code.

Changing Static Values Under Dynamic Control

Changing static values under dynamic control present a termination problem for many partial evaluators. This condition can be demonstrated utilizing the iota function previously shown in Figure 3.35 on page 129 or the factorial function in Figure 2.14

Termination Mechanism	Limitations	Advantages
Mix	First-order programs only	Computationally efficient
	Functional programs only	
	Monovariant BTA	
	No partial statics	
	Inductive variables for self recursions only	
	Can diverge on changing static values under dynamic control	
Finiteness Analysis	First-order programs only[1]	Computationally efficient
	Functional programs only	Guarantees termination
	Monovariant BTA	
	No partial statics	
	Finiteness limited to domain of structures	
	Captures decrease of a single argument, not all arguments together	
	Captures monotonic decrease, not average decrease[1]	
	Susceptible to premature termination	
Similix 1.0	(see Mix)	Handles side-effects on global variables
		(see Mix)
Similix 5.0	Monovariant BTA	Supports higher-order programs
	Lacks partial use	Supports partial statics
	Is-used analysis is eager, not lazy	Is-used analysis is an improvement over Mix and Similix 1.0
		Handles side-effects on global variables

[1] Limitation removed by later extension

Figure 4.1: Table of limitations and advantages of different termination mechanisms (offline)

Termination Mechanism	Limitations	Advantages
Fuse	Functional programs only	Dynamic analysis (online)
	Susceptible to premature termination	Diverges infrequently
Lazy Use Analysis	Functional programs only	Dynamic analysis (online)
	Large resource consumption	Supports higher-order programs
		Supports partial use
		Diverges infrequently
		Prematurely terminates infrequently

Figure 4.2: Table of limitations and advantages of different termination mechanisms (online)

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (let loop ((i 1)
                   (ans 1))
          (if (> i n)
              ans
              (loop
               (1+ i)
               (* ans i)))))))
```

Figure 4.3: Counting up factorial program (repeat of Figure 2.14)

on page 23, repeated in Figure 4.3. Partial evaluation of both of these programs utilizing a completely unknown value for `n` initiates symbolic execution of a recursion based on the `loop` function. Every iteration applies `loop` to a new set of completely computable (i.e., static) values; however, termination of the recursion is controlled by the value of the predicate of a conditional,dependent on an unknown value of `n` and a known value of `i`. The value of the dynamic expression for the predicate is not decidable during partial evaluation.

An effective termination mechanism ought to prevent divergence when presented with a recursion exhibiting changing static values under dynamic control. However, ensuring termination is complicated by the desire to produce high quality residual code for an alternative partial evaluation of the same two examples in Figures 3.35 and 4.3. Consider partial evaluation of the factorial function in Figure 4.3 applied to a specified integer. In this case, the desired result is a residual function that just returns the value of the factorial computed during partial evaluation. Computing the return value during partial evaluation requires a termination mechanism allowing the `loop` recursion to be executed to completion during partial evaluation. However, the arguments supplied to `loop` in both cases are identical, only the value of `n` differs.[2]

Premature Termination

The second example shows a case in which premature termination can occur, causing the resulting residual code to be less specialized than desired. The regular expression matcher in Figures 4.4 and 4.5 is a variant of one first utilized in an unpublished paper by Torben Mogensen to demonstrate the problem of premature termination. Partial evaluation of `match?` applied to a known regular expression and unknown input string ought to produce a residual program that is a minimal decision tree for matching the specified regular expression. No interpretive overhead or pattern representations utilized by the regular expression matcher ought to remain in the residual code. For example, partial evaluation of `match?` applied to the kleene star expression, a^* (resulting from `(make-kleene-star (make-term 'a))`), and $\perp$ ought to yield the residual program in Figure 4.6. Unfortunately, many termination mechanisms utilized by existing partial evaluators prematurely terminate symbolic execution of recursions of `match?` or `match-pattern?` before all the interpretive overhead is removed from specializations. A more detailed discussion of why this is the case and the residual code that results will be given as each alternative termination mechanism is explained.

4.1.2 Mix

Mix was one of the first offline partial evaluators. Its termination mechanism is based on a monovariant binding time analysis. Both the BTA and the other phases of the

[2]Even if the value for `n` were considered an extra virtual argument to `loop`, its value would not change from iteration to iteration in either case.

```
(define match?
  (lambda (pattern input) (match-pattern? pattern null-pattern input)))

(define match-pattern?
  (lambda (pattern rest-pattern input)
    (cond ((null-pattern? pattern)
           (match-null? rest-pattern input))
          ((term? pattern)
           (match-term? pattern rest-pattern input))
          ((kleene-star? pattern)
           (match-star? pattern rest-pattern input))
          ((concat? pattern)
           (match-concat? pattern rest-pattern input)))))

(define match-term?
  (lambda (term-pattern rest-pattern input)
    (if (and (pair? input)
             (equal? (term-symbol term-pattern) (car input)))
        (match? rest-pattern (cdr input))
        #f)))

(define match-star?
  (lambda (star-pattern rest-pattern input)
    (or (match? rest-pattern input)
        (match-pattern? (kleene-star-expr star-pattern)
                        (concat star-pattern rest-pattern)
                        input))))

(define match-concat?
  (lambda (concat-pattern rest-pattern input)
    (match-pattern? (concat-head concat-pattern)
                    (concat (concat-tail concat-pattern) rest-pattern)
                    input)))

(define match-null?
  (lambda (rest-pattern input)
    (if (null-pattern? rest-pattern)
        (null? input)
        (match? rest-pattern input))))
```

Figure 4.4: A program for matching regular expressions (part 1)

```
(define concat
  (lambda (pattern1 pattern2)
    (cond ((null-pattern? pattern1) pattern2)
          ((null-pattern? pattern2) pattern1)
          (#t (make-concat pattern1 pattern2)))))

(define make-concat
  (lambda (head tail) (list concat-id head tail)))

(define concat?
  (lambda (val) (and (pair? val) (eq? (car val) concat-id))))

(define concat-head cadr)
(define concat-tail caddr)

(define null-pattern (cons '() '()))

(define null-pattern?
  (lambda (pattern) (eq? pattern null-pattern)))

(define make-term
  (lambda (symbol) (list term-id symbol)))

(define term?
  (lambda (val) (and (pair? val) (eq? (car val) term-id))))

(define term-symbol cadr)

(define make-kleene-star
  (lambda (expr) (list kleene-star-id expr)))

(define kleene-star?
  (lambda (val) (and (pair? val) (eq? (car val) kleene-star-id))))

(define kleene-star-expr cadr)

(define concat-id 'concat)
(define term-id 'term)
(define kleene-star-id 'kleene-star)
```

Figure 4.5: A program for matching regular expressions (part 2)

```
(define match?
  (lambda (pattern input)
    (or (null? input)
        (if (and (pair? input)
                 (equal? 'a (car input)))
            (match? '(kleene-star (term-id a))
                    (cdr input))
            #f))))
```

Figure 4.6: Optimal residual code for `(match? (make-kleene-star (make-term 'a)) ⊥)`

```
(define iota
  (lambda (n)
    (loop 1 n)))

(define loop
  (lambda (i n)
    (if (= i n)
        '()
        (cons
         i
         (loop (1+ i) n)))))
```

Figure 4.7: First-order iota function

Mix partial evaluator are described as a prelude to discussing the pros and cons of Mix's termination mechanism.

Phases in the Mix Partial Evaluator

There are five phases in the Mix partial evaluator: binding time analysis, program annotation, function specialization, call graph analysis, and call unfolding and reduction. (The descriptions of the purposes and operations performed in each of these phases come from [28].) The iota function will be used as a working example for presenting how Mix operates. However, as Mix only handles a first-order language, the version of iota in Figure 3.35 is replaced by the equivalent first-order version of iota appearing in Figure 4.7.

Binding Time Analysis Phase: Mix utilizes a monovariant BTA, meaning it creates one, and only one, binding time annotation for each function in a source program. Binding time analysis is implemented as an abstract interpretation of the input program utilizing the two element domain composed of the values static and dynamic. As explained in Section 2.2.1, static means a value is always known during partial evaluation and dynamic means the value may or may not be known.

The rules for performing BTA in Mix are very simple. Initially, all variables are annotated as static, except those forming the initial function application describing the inputs for which the program is to be partially evaluated. These inputs are annotated as either static or dynamic based on the input specification. Abstract interpretation proceeds with symbolic evaluation of the body of the initial function application. Any time a dynamic value is computed for a variable previously annotated as static, the variable is changed to be dynamic and all expressions whose values are dependent on the variable are recomputed.

BTA is guaranteed to terminate because there are only a finite number of variables in any source program, each starts out annotated static and can only change to dynamic once, and the only operation that causes more analysis to be performed is the changing of a variable's annotation from a static to dynamic.

The result of each expression is assigned either a static or dynamic annotation as follows. Known constants are by definition static. Variables' annotations are maintained by the abstract interpretation. All applications and control flow operators yield dynamic results unless all of their arguments are static.

For partial evaluation of iota applied to an unknown value, BTA begins with the application of `iota` to a dynamic value. The argument `n` of `iota` is initially annotated dynamic, and all other variables are annotated static. As a result of the application of `iota` to the dynamic value, an application of `loop` to a static (constant) value and a dynamic value is performed. Since `i` is static in `loop`, `(1+ i)` returns a static value. This means the recursive call to `loop` is made using a static and a dynamic value. Since this is the identical set of annotations utilized in the initial application, BTA has completed annotation of all the variables used in iota. The result of BTA appears in Figure 4.8 in which static variables and expressions have a superscript of `S` and dynamic variables and expressions a superscript of `D`.

```
(define iota
  (lambda (n$^D$)
    ($^D$loop 1$^S$ n$^D$)))

(define loop
  (lambda (i$^S$ n$^D$)
    ($^D$if ($^D$= i$^S$ n$^D$)
          '()$^S$
          ($^D$cons
            i$^S$
            ($^D$loop ($^S$1+ i$^S$) n$^D$)))))
```

Figure 4.8: BTA of iota applied to a dynamic value

Program Annotation Phase: Program annotation labels each function application and invocation of a control flow operator as either static or dynamic based on the results of BTA. Primitive applications are labeled as yielding static results if and only if all of their arguments are static. Control flow operators are labeled based on the arguments in which they are strict. For example, a conditional is labeled as static if and only if its predicate is static.

Applications of nonprimitive functions are annotated as unfoldable or residual. These annotations determine whether the partial evaluator terminates. The bodies of functions applied in unfoldable calls are symbolically evaluated during the function specialization phase. Infinite unfolding yields divergence.

A nonprimitive application is labeled as unfoldable in only two cases. The first is if all of its arguments are static. Clearly if all the arguments are static then the partial evaluator ought to be able to compute the result of the application utilizing the known argument values. The second case is when an application is self recursive (i.e., the function being applied is the function whose body is being analyzed), there is at least one *inductive argument*, and all other static arguments remain unchanged between the initial application and the recursive call. An inductive argument is one in which the value utilized in the recursive call is a proper substructure of the value supplied to the initial call. For example, the `lst` argument to `loop` in Figure 2.7 on page 15, repeated in Figure 4.9, is a recursive argument since the recursive application of `loop` always utilizes the cdr of the previous value of `lst`.

```
(define length
  (lambda (lst)
    (loop lst 0)))

(define loop
  (lambda (lst ans)
    (if (null? lst)
        ans
        (loop (cdr lst) (1+ ans)))))
```

Figure 4.9: A function for computing the length of lists (repeat of Figure 2.7)

Returning to the iota example, the application of `loop` in `iota` is annotated residual because it is an application of a nonprimitive function to a dynamic argument. The primitive application of `1+` in `loop` is static since it is applied to a single static argument; whereas, the primitive application of `=` is annotated dynamic since one of its arguments is dynamic. The application of `cons` is annotated dynamic on the assumption that the cons primitive is strict in its dynamic argument. The recursive application of the nonprimitive function `loop` is annotated residual because it is a direct recursive call, but none of the static arguments is an inductive variable. Finally, the control flow operator `if` is annotated dynamic since its predicate is dynamic. The result of the annotation phase appears in Figure 4.10 in which subscripts of S, D, R, and U have been utilized to represent static, dynamic, residual, and unfoldable, respectively.

Function Specialization Phase: The function specialization phase creates specializations of functions utilizing symbolic evaluation. The algorithm is based on a work list of specializations still needing to be created and a table of completed specializations. The work list begins with a single entry for the initial application of the function to be specialized applied to the input specification. The algorithm proceeds by removing items from the work list and creating specializations for them until the work list is empty. Each time an item is removed from the worklist, a corresponding entry is created in the table of specializations.

In order to create a specialization for an application from the work list, the function

```
(define iota
  (lambda (n$^D$)
    (${}^D_R$loop 1$^S$ n$^D$)))

(define loop
  (lambda (i$^S$ n$^D$)
    (${}^D_D$if (${}^D_D$= i$^S$ n$^D$)
          '()$^S$
          (${}^D_D$cons
            i$^S$
            (${}^D_R$loop (${}^S_S$1+ i$^S$) n$^D$)))))
```

Figure 4.10: Result of the annotation phase of Mix for iota applied to a dynamic value

specializer performs symbolic evaluation of the function's body utilizing the values for its static arguments. The static results of static primitive applications are computed, while the results of dynamic primitive applications cannot be determined so residual applications are created. Static control flow operators are removed and replaced by the results of symbolic evaluation of the appropriate flow of control. For dynamic flows of control, residual control flow operations remain and all possible flows of control are analyzed. Finally, symbolic evaluation of unfoldable, nonprimitive applications is performed precisely as is currently being described.

The only remaining case is nonprimitive applications annotated as residual. Residual applications are compared with those in the table of specializations already created. If no specialization already exists for an application of the given function to identical values for the static arguments, then a specification of the specialization to be created is placed on the work list. Whether the needed specialization already exists or not, a residual application applying either the existing or new specialization is created in this case. Although Jones et. al. do not state in [28] how symbolic evaluation proceeds after the residual application is created, it is my understanding the analysis proceeds with symbolic execution of the continuation of the residual application applied to the completely unknown value.

Once again continuing the iota example, function specialization of iota applied to an unknown argument begins by placing the application (${}^D_R$`iota` $\perp^D$) on the work list. When this application is removed from the work list, it immediately leads to symbolic evaluation of the expression ($^{D}_{R}$`loop 1`^{S} $\perp^{D}$). Since this is a residual application, the application of `loop` to the static value `1` is placed on the work list. Later, ($^{D}_{R}$`loop 1`^{S} $\perp^{D}$) is removed from the work list and placed in the table of specializations. The application of the primitive function `=` in the body of `loop` is dynamic, so it produces an unknown result. This is reflected by the fact that the control flow operator `if` was annotated dynamic. The consequent of the conditional is a simple constant and is therefore not particularly interesting. Symbolic evaluation of the alternative begins with the application of the primitive function `1+` to the static value `1`. This static application is evaluated to yield `2`. Symbolic evaluation is then ready to handle the expression ($^{D}_{R}$`loop 2`^{S} $\perp^{D}$). Since ($^{D}_{R}$`loop 2`^{S} $\perp^{D}$) is a residual application and no entry appears in the table of specializations for `loop` applied to the static value `2`, this application is placed on the work list. When specialization of ($^{D}_{R}$`loop 1`^{S} $\perp^{D}$) completes. ($^{D}_{R}$`loop 2`^{S} $\perp^{D}$) is removed from the worklist. Symbolic evaluation of ($^{D}_{R}$`loop 2`^{S} $\perp^{D}$) leads to evaluation of the expression ($^{S}_{S}$`1+ 1`^{S}) yielding the value `3`. This in turn leads to the placing of ($^{D}_{R}$`loop 3`^{S} $\perp^{D}$) on the work list. The process of investigating applications of `loop` to each of the positive integers proceeds ad infinitum, leading to divergence. As demonstrated, Mix is susceptible to divergence when processing programs containing changing static values under dynamic control.

Call Graph Analysis Phase: The purpose of the call graph phase is to detect recursions in the specializations produced by the function specialization phase. A standard call graph is produced for the program resulting from the function specialization phase. A depth first walk is performed of the call graph starting with the node representing the initial application of the program applied to the input specification, to be referred to as the *root* node. Each time a node is reached for a second time on a single path from the root node, a recursive loop has been detected, and the node is marked as a *cutpoint* in order to break the recursion.

In order to proceed with the iota example, assume BTA annotated both arguments of `loop` dynamic, instead of one static and one dynamic. Then, function specialization would only have created a single specialization of `loop` applied to two dynamic values. This yields a very simple call graph for iota applied to an unknown value. There are two nodes for the two specializations: (^D_R`iota` $\perp^D$) and (^D_R`loop` $\perp^D$ $\perp^D$). There is one arc from (^D_R`iota` $\perp^D$) to (^D_R`loop` $\perp^D$ $\perp^D$), and another arc from (^D_R`loop` $\perp^D$ $\perp^D$) to itself. Call graph analysis annotates (^D_R`loop` $\perp^D$ $\perp^D$) as a cutpoint since the arc from (^D_R`loop` $\perp^D$ $\perp^D$) to itself forms a loop in the call graph.

Call Unfolding and Reduction Phase: The final phase performs further unfolding (i.e., inlining) of function applications. All function applications are unfolded unless one of three problems might result: infinite unfolding, duplication of a function call, or duplication of a complex expression. Protection against infinite unfolding is achieved by choosing not to unfold any applications denoted as cutpoints by the call graph analysis phase. Duplication of function applications is avoided by checking the argument values of each application to determine which arguments are pieces of code containing function applications and only performing unfolding if those arguments are used at most once in the body of the function being applied.[3] Finally, avoidance of duplication of complex argument expressions is handled identically to the case of function call duplication.

No unfolding is possible for the continuation of the iota example. The only residual calls are the ones to `loop`. Since `loop` was annotated as a cut point, it is not unfoldable. However, imagine the function specialization phase had produced two specializations of `loop`, one for even values of the first argument and the other for odd values, to be referred to as `loop-e` and `loop-o`, respectively. The resulting specializations would look something like those in Figure 4.11. The resulting call graph would contain three nodes: one for `iota`, one for `loop-e`, and one for `loop-o`. In addition to the arc from `iota` to `loop-o`, there would be an arc from `loop-o` to `loop-e` and one from `loop-e` to `loop-o`. Call graph analysis would annotate `loop-o` as a cutpoint. Now, the call to `loop-e` in the body of `loop-o` would be unfoldable during the call unfolding phase, assuming the duplicated expression `(1+ i)` is not considered too expensive to be replicated. However, applications of `loop-o` would not be unfolded since `loop-o` is a cutpoint. The resulting residual program would look something like the one shown in Figure 4.12.

[3]For reasons that are not specified, Mix does not address the problem of duplicating function applications through the creation of a temporary location to store the result of the potentially

```
(define iota
  (lambda (n)
    (loop-o 1 n)))

(define loop-o
  (lambda (i n)
    (if (= i n)
        '()
        (cons
         i
         (loop-e (1+ i) n)))))

(define loop-e
  (lambda (i n)
    (if (= i n)
        '()
        (cons
         i
         (loop-o (1+ i) n)))))
```

Figure 4.11: Hypothetical function specialization from Mix

```
(define iota
  (lambda (n)
    (loop-o 1 n)))

(define loop-o
  (lambda (i n)
    (if (= i n)
        '()
        (cons
         i
         (if (= (1+ i) n)
             '()
             (cons
              (1+ i)
              (loop-o (1+ (1+ i)) n)))))))
```

Figure 4.12: Hypothetical residual code from Mix for iota

Advantages and Disadvantages of Mix's Termination Mechanism

The termination mechanism utilized by Mix has both advantages and disadvantages when compared with a termination mechanism based on lazy use analysis. First the limitations with the Mix approach will be presented and then its advantages are discussed.

Limitations of Mix

Mix's termination mechanism has five significant limitations. It only supports a first-order source language, uses a monovariant BTA, does not support partial statics in its BTA, only handles inductive variables in self recursions, and fails to terminate on one of the motivating examples. Each of these limitations is discussed in order.

First-Order Source Language: Mix's termination mechanism differs from lazy use analysis being only applicable to first-order source languages. Lazy use analysis has as one of its goals the ability to handle higher-order languages and the programming paradigms commonly utilized by programmers using higher-order languages such as Scheme. While later research has developed versions of several passes of Mix that handle higher-order languages (e.g., higher-order BTA [13]), other aspects of the algorithm require major reworking in order to apply Mix's approach to termination to higher-order languages.

Monovariant BTA: Utilization of a monovariant BTA causes Mix to produce suboptimal code in some cases. For example, consider the function for computing binomial coefficients in Figure 4.13. Partial evaluation by Mix of `binomial-coeff` for a known value of `n` and an unknown value of `k` yields unnecessarily inefficient residual code. `binomial-coeff` applies `fact` to two different types of arguments. In the first application, the argument is a known constant value during specialization. In the second and third applications, the value of the argument to `fact` is not known until runtime. Ideally, the value of the first factorial ought to be computed during partial evaluation and the constant result placed in the residual code, while the other two factorial computations must be performed at runtime. However, monovariant BTA can only label

duplicated function application.

```
(define binomial-coeff
  (lambda (n k)
    (/ (fact n)
       (* (fact k) (fact (- n k))))))
```

Figure 4.13: Function to compute binomial coefficients

factorial with one set of annotations. Since the argument is static in one location and dynamic in the other two locations, the annotation created for the argument to `fact` is dynamic. This prevents computing the result of factorial applied to the static value during partial evaluation.

Lazy use analysis is inherently polyvariant. A partial evaluator utilizing a termination mechanism based on lazy use analysis computes factorial of the static value in `binomial-coeff` during partial evaluation. A potential specialization of `fact` for an unknown input is created for the other two application.

Limitations in Mix's termination mechanism resulting from the monovariance of its BTA are easily corrected by replacing the monovariant BTA with a polyvariant one. Therefore, the monovariance of the BTA utilized by Mix ought not to be considered a significant limitation, as polyvariant BTAs are now available (e.g., [12]).

Unable to Represent Partially Known Values: Mix's BTA can only annotate values as static or dynamic. It lacks the ability to characterize a value as partially known and partially unknown. The ramifications of this limitation are demonstrated by a simple data abstraction for modeling stores shown in Figure 4.14, similar to one used by Ruf in [37]. The store is represented as an association list of name/value pairs. The implementation is functional so operations adding values to the store return a new store.

Consider the strange version of the identity function implemented using a store shown in Figure 4.15. Specialization of `store-and-lookup` on an unknown input ought to yield the traditional identity function `(lambda (value) value)`; however, Mix does not produce this simple residual program because its monovariant BTA is unable to represent that the entry placed in the store has a known, static name and an unknown, dynamic value. As a result, the entire store passed to `lookup-in-store`

```
(define in-store?
  (lambda (name store)
    (cond ((null? store) #f)
          ((eq? name (entry-name (first-entry store))) #t)
          (#t (in-store? name (rest-entrys store))))))

(define lookup-in-store
  (lambda (name store)
    (if (eq? name (entry-name (first-entry store)))
        (entry-value (first-entry store))
        (lookup-in-store name (rest-entrys store)))))

(define update-store
  (lambda (store name new-value)
    (let ((binding (first-entry store)))
      (if (eq? (entry-name binding) name)
          (cons (cons name new-value) (rest-entrys store))
          (cons binding (update-store (rest-entrys store) name new-value))))))

(define add-to-store
  (lambda (store name new-value)
    (cons (cons name new-value) store)))

(define empty-store '())

(define first-entry
  (lambda (store)
    (car store)))

(define rest-entrys
  (lambda (store)
    (cdr store)))

(define entry-name
  (lambda (entry)
    (car entry)))

(define entry-value
  (lambda (entry)
    (cdr entry)))
```

Figure 4.14: Functions to access and update (functionally) a store

```
(define store-and-lookup
  (lambda (value)
    (lookup-in-store
     'a
     (add-to-store empty-store 'a value))))
```

Figure 4.15: Identity function based on a store

is annotated dynamic and virtually no optimization is performed.

Mix's solution to the problem of partially static data structures is to require the user to rewrite their program to separate the static portions of data structures from the dynamic ones. Mix's BTA is then able to better analyze the program and to produce better residual code. Subsequent to Mix, *partially static* BTAs were developed by Mogensen [21] and Consel [12]. These BTAs are able to annotate values as not only static or dynamic, but, in the case of aggregates, as being composed of some known pieces annotated static and other unknown pieces annotated dynamic.

Lazy use analysis by its very nature is able to represent that some parts of an aggregate are known and other parts are unknown, as is apparent from the domains in Figure 2.22 on page 31, repeated as Figure 4.16. However, lazy use analysis goes beyond the realm of partially static structures. Lazy use analysis incorporates the more more general concept of *partial information* of which partially static structures are just one example. Much as an aggregate is a union of a number of separable pieces of information, each of which can be known or unknown during partial evaluation, scalars are also composed of many separable pieces of information. The value `3` is a member of the sets `{3}`, integers, and numbers, each based on successively less information about `3`. Elements of the domains in Figure 4.16 are able to represent both that `3` is a member of the set `{3}` and that it is an integer. The ability to represent the partial information that a value is an integer, without knowing its precise numerical value, is a form of partial information not captured by partially static BTA.

Inductive Variables Limited to Self Recursions: In addition to annotating as unfoldable function applications for which all of the arguments are static, self-recursive applications with inductive variables are also annotated as unfoldable by

Int	$=$	$\mathtt{0} + \pm\mathtt{1} + \pm\mathtt{2} + \cdots$	integers
		$\bot_{Int}$	unspecified integer
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
		$\bot_{Bool}$	unspecified boolean
Sym	$=$	$\mathtt{'a} + \mathtt{'b} + \cdots$	symbols
		$\bot_{Sym}$	unspecified symbol
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\bot_{Pair} \equiv \bot_{PEval} \times \bot_{PEval}$	unspecified pair
$Closure$	$=$	$Lambda \times Env$	closure values
		$\bot_{Clos}$	unspecified closure
Env	$=$	$(Id \to PEval)^{\star}$	environments
$Kval$	$=$	$Int + Bool + Nil + Pair + Closure$	known values
$Bots$	$=$	$\bot_{Int} + \bot_{Bool} + \bot_{Pair} + \bot_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \bot_{PEval}$	partial evaluation values
		$\bot_{PEval}$	unspecified value

Figure 4.16: Value domains for partial evaluation of a pure subset of Scheme (repeat of Figure 2.22)

Mix. The limitation of the inductive variable heuristic to self-recursive functions is fairly significant. Any recursion having a static argument that is an inductive variables may be safely unfolded by a partial evaluator, even if the recursion is not based on a self-recursive function. Furthermore, inductive variables are just one special case of a monotonic progression through the elements of a well-founded set. It is safe to unfold applications any time a variable is maintaining a monotonic progression through the elements of a well-founded set. The key is being able to detect that this is happening.

Holst's Finiteness Analysis [26], discussed in the next section, is an improvement on the inductive variables of Mix. However, sticking for the time being to the termination mechanism utilized by Mix, lazy use analysis offers a better solution to inductive variables than Mix. Any time Mix unfolds a self-recursive application due to an inductive variable, lazy use analysis creates a new potential specialization for the application, so long as the information in the inductive variable is utilized by the recursive application. Lazy use analysis allows for unfolding of recursions any time Mix allows for unfolding and the unfolding is effectively improving the quality of the residual code. Furthermore, lazy use analysis enables unfolding for all recursions, not just self recursions.

Divergence: The final limitation of Mix's termination mechanism is divergence in some cases of changing static values under dynamic control. Whereas lazy use analysis terminates when partially evaluating the iota function in Figure 4.7 on page 141, Mix diverges. In this regard, lazy use analysis is clearly a superior termination mechanism to the one utilized by Mix.

Advantages of Mix

The major advantage of Mix's termination mechanism over lazy use analysis is efficiency. All five phases of Mix operate quite rapidly and use a small amount of memory in comparison with lazy use analysis. As a result of the large resource consumption of lazy use analysis, it has not been applied to anything but very small programs. Mix, on the other hand, has been utilized to optimize some reasonable size programs. Whereas Mix can be utilized to optimize some real programs, lazy use analysis is unable to handle real programs as currently implemented.

4.1.3 Finiteness Analysis

Holst's finiteness analysis, as presented in [26], is an addition to an offline partial evaluator for a first order language that modifies the BTA annotations of applications in order to insure termination of partial evaluation of convergent input programs. Termination is guaranteed by systematically converting static annotations to dynamic until finiteness analysis can prove only a finite number of applications of functions to different sets of static arguments will take place during the specialization phase of partial evaluation. The proof of finiteness is based on modeling whether the size of different arguments is growing or shrinking during recursions. The intuitive basis of the algorithm is that infinite looping can only take place during partial evaluation if some argument grows an unbounded number of times.

This presentation of finiteness analysis proceeds with a more detailed discussion of the principles behind the analysis. Once the concepts have been illuminated, how the analysis operates in practice is explained. This is followed by a discussion of the limitations and advantages of finiteness analysis. The section concludes with a brief presentation of two extensions to finiteness analysis.

A complete presentation of finiteness analysis requires building up a vocabulary of terms and concepts utilized in formulating the analysis. Therefore, this discussion proceeds through a series of definitions, culminating in the final theorem central to finiteness analysis.

A program is said to be *quasi-terminating* if it enters at most a finite number of distinct states. A quasi-terminating program does not necessarily terminate since it might loop infinitely amongst some subset of the finite number of states. However, partial evaluation of a quasi-terminating input program is guaranteed to terminate, as explained below.

During the specialization phase of an offline partial evaluator, the functions applied and the static arguments to which they are applied serve as a reasonable representation of the states of the analysis. If an input program is quasi-terminating, in order for partial evaluation of the program to diverge, the partial evaluator must pass through at least one state an infinite number of times. However, any time an offline partial evaluator reaches an identical state (i.e., the same function applied to identical static arguments) a second time, symbolic execution of that flow of control is terminated and a residual application of the specialization for that state is created. This prevents an infinite loop through the identical set of states and thereby guarantees termination.

The next important concept in finiteness analysis is the size of values. Each argument of a function application can be compared with each of the arguments of the preceding application in the execution sequence of a program. Each of the arguments in the later application can be classified as *decreasing*, *constant*, or *increasing* in size in comparison to each of the arguments of the former. An argument is said to be decreasing in comparison with a previous argument if it is strictly smaller than that argument; it is constant, if it is identical to it; and, it is *weakly decreasing*, if it is either smaller or identical. Finally, finiteness analysis is conservative; so, any time an argument cannot be shown to be decreasing, constant, or weakly decreasing, it is considered to be increasing.

All of the above definitions are based on a concept of size. While there are many domains for which a concept of size is meaningful, finiteness analysis as implemented only seems to apply the concept of size to the domain of structures composed of cons cells.[4] For the domain of cons cells, the concept of size ought to be intuitively obvious. Application of finiteness analysis to other domains of values (e.g., numbers) probably ran into problems with the *boundedness* criteria, to be presented once the current discussion of size is completed.

Because finiteness analysis is only able to assign relative sizes to structures composed of cons cells, the analysis is unable to determine whether other data types are increasing or decreasing in size. One consequence of this limitation is that finiteness analysis unnecessarily reannotates some static parameters as dynamic, leading to premature termination. For example, partial evaluation of the factorial function in Figure 2.14 on page 23 applied to a known, constant input yields a recursive residual program, rather than a program just returning the result computed during partial evaluation. This happens because finiteness analysis reannotates one of the arguments to `loop` as dynamic in order for it to be able to prove the loop terminates.

Once the concept of size has been defined across a single function application boundary, it can be generalized for multiple applications by defining how size changes are combined. Combining two decreases, a decrease and a constant, or a decease and a weak decrease all produce a decrease. Combining two constants produces a constant. Combining two weak decreases, a weak decrease and a decrease, or a week decrease and a constant all produce a weak decrease. All other combinations are conservatively approximated as an increase.

The important case for finiteness analysis is changes in size of arguments during recursions. Based on the above definitions for decreasing, constant, and increasing arguments, it is possible to characterize changes in the size of each argument between an initial application of a function and a later recursive application of the same function. These characterizations of changes in size of each argument form the basis of the central theorem of finiteness analysis.

The final concept behind finiteness analysis is that of *boundedness*. An argument position of a function is said to be bounded if there are only a finite number of

[4]This limitation of the implementation is inferred from Holst's paper and is not explicitly stated anywhere.

values the corresponding formal parameter of the function can assume during partial evaluation of an input program. If it cannot be proven an argument is bounded, it is considered to be *unbounded*.

The central theorem of finiteness analysis states that any recursion with an increasing argument position is quasi-terminating so long as there is another argument position that is bounded and decreasing. Intuitively, if one argument of a recursion can assume at most a bounded number of values and will assume values in a strictly decreasing fashion, the recursion must eventually terminate. Otherwise, the recursion would run out of successively smaller values for the argument to assume. A consequence of this theorem is that termination of partial evaluation can be guaranteed by reannotating static increasing argument positions as dynamic whenever there is no other argument position of the same application that is bounded and decreasing. This suffices to guarantee termination since any recursion with no increasing argument positions is quasi-terminating.

Limitations of Finiteness Analysis

There are five primary limitations of finiteness analysis as originally developed: applicability only to first-order input languages, inability to work in conjunction with partially static BTA's, implementation only for domains of structures, inability to consider whether a collective set of arguments is decreasing in size even though a single argument may be temporarily increasing, and inability to detect arguments that temporarily increase in size but on the average are decreasing. Each of these limitations is discussed in order. Particular emphasis is given to the third and fourth limitations that lead to premature termination and the generation of poor residual code for partial evaluation of iota and the regular expression matcher, respectively.

First-Order Source Language: Applicability only to first-order source languages was actually only a limitation of the initial design of finiteness analysis. Andersen and Holst in [3] extended finiteness analysis to handle higher-order languages.

Incompatibility with Partially Static BTA: Finiteness analysis is not compatible with partially static BTA. In order to achieve the benefits of guaranteed termination yielded by finiteness analysis, a partial evaluator must give up the advantages of partial statics. In particular, a partial evaluator loses the ability to form specializations based on the static portions of arguments when other parts of the same data structures are dynamic. A BTA unable to represent partial statics is forced to annotate the entire structure as dynamic. The consequence is less specialized and therefore often less highly optimized residual code.

Only Implemented for Domains of Structures: While it is not explicitly stated anywhere in [26], it seems the concept of size in finiteness analysis has only been applied to domains of structures. This assumption is reinforced by the fact that all of the examples in Holst's paper requiring analysis of numeric domains are presented using unary numbers represented by lists whose lengths are the values being represented. Unfortunately, this encoding does not solve the general problem of analysis of numeric domains, not even the subdomain of non-negative integers.

Since the concept of size has only been applied to domains of structures, any time an argument assumes two different non-structure values during a recursion it must be annotated as increasing. Resulting reannotation of static arguments as dynamic can lead to premature termination of recursions based on data types other then structures. For example, partial evaluation of the iota function in Figure 4.7 on page 141 applied to a known, constant value is desired to produce straight line residual code to generate the return value. However, finiteness analysis yields a less desirable result.

The argument `i` of `loop` is annotated as increasing by finiteness analysis. Since the argument `n` of `loop` is annotated constant, the static annotation of `i` is converted to dynamic by finiteness analysis. The net result is that a single, recursive specialization of `loop` is created, specialized for the constant value of `n`. This is clearly a highly unoptimized result, far poorer than the one produced by lazy use analysis.

One Argument Increasing in Size, Collection of Arguments Decreasing: Another limitation of finiteness analysis results from its concept of size being limited to single argument positions. If all of the arguments to a function in aggregate are getting smaller, termination is guaranteed, even though a single argument may be increasing in size and there is no bounded, decreasing argument. The inability of finiteness analysis to recognize this condition results in poor residual code when performing partial evaluation of the regular expression matcher in Figures 4.4 and 4.5 on pages 139 and 140, repeated in part in Figure 4.17.

Consider partial evaluation of the regular expression matcher applied to a static pattern and a dynamic input string. One flow of control analyzed by finiteness analysis for a kleene star pattern proceeds through applications of `match-pattern?`, `match-star?`, `match?` and then `match-pattern?` again. `Input` remains constant in size around this recursion; `pattern` is annotated as increasing; and, `rest-pattern` is annotated as increasing.

Since `input` is just passed from one application to the next it is obvious that it must be constant. The simplest means of explaining why `pattern` is annotated as increasing is through an example demonstrating a case in which `pattern` increases in size. Assume the first application of `match-pattern?` starts with `pattern` bound to a simple kleene star expression like a^* and `rest-pattern` bound to a large, complex expression. In this case, `match?` is applied to the large complex rest pattern and the original input. As a consequence, `match-pattern?` is applied to the large complex expression, a null pattern, and the original input string. Since the large, complex expression is larger in size than a^*, `pattern` has increased in size and therefore it must be annotated as increasing if finiteness analysis is to produce correct results.

Since `pattern` is annotated as increasing, finiteness analysis must reannotated `pattern` as dynamic unless some other argument is bounded and decreasing. Since `input` is constant, the only possibility is `rest-pattern`. However, another simple example demonstrates that `rest-pattern` can remain constant in size through the recursion being considered and therefore cannot correctly be annotated as decreasing.

Assume, `rest-pattern` starts off as the null pattern. `Match?` is then applied to the null pattern and the original input. The recursive application of `match-pattern?` is made to two null patterns and the original input. Since one null pattern cannot be smaller than another null pattern, `pattern` clearly cannot be annotated as decreasing.

Finiteness analysis reannotates the `pattern` and `rest-pattern` arguments to `match-pattern?` as dynamic. Based on these reannotations, all of the variables in the regular expression matcher end up annotated dynamic. Clearly, partial evaluation of the regular expression matcher based on completely dynamic annotations yields residual code roughly equivalent to the source program and far less optimized than

```
(define match?
  (lambda (pattern input) (match-pattern? pattern null-pattern input)))

(define match-pattern?
  (lambda (pattern rest-pattern input)
    (cond ((null-pattern? pattern)
           (match-null? rest-pattern input))
          ((term? pattern)
           (match-term? pattern rest-pattern input))
          ((kleene-star? pattern)
           (match-star? pattern rest-pattern input))
          ((concat? pattern)
           (match-concat? pattern rest-pattern input)))))

(define match-term?
  (lambda (term-pattern rest-pattern input)
    (if (and (pair? input)
             (equal? (term-symbol term-pattern) (car input)))
        (match? rest-pattern (cdr input))
        #f)))

(define match-star?
  (lambda (star-pattern rest-pattern input)
    (or (match? rest-pattern input)
        (match-pattern? (kleene-star-expr star-pattern)
                        (concat star-pattern rest-pattern)
                        input))))

(define match-concat?
  (lambda (concat-pattern rest-pattern input)
    (match-pattern? (concat-head concat-pattern)
                    (concat (concat-tail concat-pattern) rest-pattern)
                    input)))

(define match-null?
  (lambda (rest-pattern input)
    (if (null-pattern? rest-pattern)
        (null? input)
        (match? rest-pattern input))))
```

Figure 4.17: A program for matching regular expressions (repeat of Figure 4.4)

```
(define match?
  (lambda (pattern input)
    (or (null? input)
        (if (and (pair? input)
                 (equal? 'a (car input)))
            (match? '(kleene-star (term-id a))
                    (cdr input))
            #f))))
```

Figure 4.18: Optimal residual code for `(match? (make-kleene-star (make-term 'a)) ⊥)` (repeat of Figure 4.6)

the desired result produced by lazy use Analysis shown in Figure 4.6 on page 141, repeated in Figure 4.18.

One interesting fact to note about the regular expression matcher is that if all the arguments of each function are considered as a group, their collective size decreases with each recursive application made by the program. In every instance, either a symbol of the input stream is consumed in the matching process or the joint size of the two pieces of the pattern is reduced. Finiteness analysis fails to capture this form of reduction in size and consequently generates poor residual code for the regular expression matcher. It trades off guaranteed termination against residual code quality.

Temporary Increase in Size, But Average Decrease: The last limitation is the inability of finiteness analysis to recognize a condition in which an argument temporarily increases in size, but on the average is continually decreasing. For example, consider the code in Figure 4.19. Each call to `increase` increases the size of the argument by one cons cell; but, each call to `decrease` reduces the size of the argument by two cons cells. Therefore, each recursive application of `increase` utilizes an argument that is one cons cell smaller than the one in the previous application. However, finiteness analysis annotates `arg` as increasing in both function definitions since it does not represent the amount by which something has increased or decreased. It is therefore unable to recognize that an increase plus a larger decrease represents a net decrease in size.

Das and Reps in [17] extend Holst's original work in finiteness analysis, a well

```
(define increase
  (lambda (arg)
    (if (not (null? arg))
        (decrease (cons 1 arg)))))

(define decrease
  (lambda (arg)
    (increase (cddr arg))))
```

Figure 4.19: Argument temporarily increases in size, but on average continually decreases

as the work of Andersen and Holst [3] and Glenstrup and Jones [22], in order to handle the problem of arguments temporarily growing in size, but having a net decrease by the time a recursive application is performed. The Das and Reps approach also addresses a subtle condition in the original Holst system relating to symbolic execution of error conditions during partial evaluation. In all other regards, Das and Reps's extensions to finiteness analysis share the termination properties of the original analysis.

Advantages of Finiteness Analysis

The major advantage of finiteness analysis is guaranteed termination of partial evaluation for all terminating source programs. It is the only termination mechanism presented herein able to make this claim. Whether degradation in residual code quality in some cases resulting from premature termination is a good trade-off against guaranteed termination continues to be a source of some debate in the partial evaluation community.

4.1.4 Similix

There have been five major releases of Similix to date. As the Similix system has evolved, its termination mechanism has changed. The termination mechanisms of two representative versions of Similix on which papers have been published will be discussed herein. The limitations and advantages of the latter version of Similix are then discussed

Similix 1.0

Similix 1.0 is a three phase, offline partial evaluator for a first-order subset of Scheme supporting side effects only on global variables. The three phases of the partial evaluator are preprocessing of the source (including binding time analysis), specialization, and post unfolding. The only portion of the preprocessing impacting termination in any significant way is the BTA, which is monovariant. There are no termination issues in the post unfolding. Consequently, this discussion concentrates on termination of the specialization phase as presented in [8].

One major difference between Similix and other partial evaluators is its selection of different specialization points. Whereas most partial evaluators utilize function applications as specialization points, Similix uses dynamic conditionals as its specialization points. This has a small and not terribly significant impact on the termination properties of Similix, but a significant impact on its performance.

The first step in the specialization phase is the introduction of dummy procedure calls at the point of each dynamic conditional. These dummy applications act as specialization points and also function as the points at which termination decisions are made. Each dynamic conditional becomes the body of a new function. The arguments to the function are all of the free variables referenced in the predicate, consequent, or alternative of the conditional. The arguments of the new function are annotated as static or dynamic based on the annotations of the corresponding variables in the source program. The new function is applied to the appropriate values at the location in the input program at which the conditional previously appeared.

During the specialization phase Similix operates in a fashion very similar to Mix. However, specialization and termination decisions are only made at application sites for the newly introduced applications. Each time symbolic execution reaches one of the applications introduced for dynamic conditionals, the values of all of the static arguments of the application are compared with the corresponding values of any previous applications of the same function. If no matching set of arguments is found then creation of a new specialization is initiated. Otherwise, a specialization already exists for the specified static arguments so symbolic execution of the thread is effectively terminated with the application.

The termination properties of Similix differ from those of Mix in two ways. First, the Similix 1.0 termination mechanism does not include the second order heuristic of Mix for self-recursive procedures with an argument that is shrinking in size. As a result, Similix terminates some recursions more rapidly than Mix, producing less specializations. However, the Mix heuristic is guaranteed not to effect whether the partial evaluator eventually terminates for any given input a program.

Similix diverges for some inputs on which Mix converges. Since the only specialization points at which termination decisions are made are dynamic conditionals, Similix can diverge specializing statically controlled loops that propagate at least one dynamic value. This is a special case of what Weise and I called *hidden divergence* in [31]. That Mix terminates when processing some programs with hidden divergences, while Similix 1.0 does not, is not terribly significant.

The termination mechanisms of Mix and Similix 1.0 pretty much share all the same limitations and advantages. Just like Mix, Similix 1.0 diverges performing partial evaluation of iota applied to an unknown argument. However, both handle the regular expression matcher example as desired. The one main advantage of Similix 1.0 over Mix is that the use of specialization points introduced for dynamic conditionals yields a more efficient partial evaluator.

Similix 5.0

Several significant changes were made in Similix between version 1.0 and version 5.0. Version 2.0 extended version 1.0 with higher-order functions using a combination of Sestoft's closure analysis and Reynolds's defunctionalization [4, 6, 41, 35]. Version 3.0 extended version 2.0 with partially static values [5]. The specializer of version 4.0 is continuation-based [7, 14, 16]. The analyses of version 5.0 are constraint-based [9, 25]. This description of version 5.0 is based on [9]. It concentrates on a new *is-used* analysis added to version 5.0 that modifies the binding times of some terms and results in improved termination properties.

The concept behind is-used analysis is to identify static values and expressions not contributing to making control flow decisions. Static expressions not used in this sense are reannotated as dynamic prior to the specialization phase of partial evaluation. Conversion of annotations for some arguments of functions from static to dynamic can result in termination of the specialization phase of partial evaluation when it previously diverged. Two applications differing only in the value of an argument reannotated as dynamic are treated as equivalent after the reannotation.

Similix 5.0 performs is-used analysis utilizing a constraint satisfaction system. Is-used constraints are created for the following four types of expressions. Static constants are all annotated as used. If the predicate of a conditional is static, then it is annotated as used. Static applications of Similix's type predicates for its record types are annotated as used. Finally, static applications of what Similix calls *transparent* (always reducible) primitive functions are annotated as used.

The last case requires some explanation. Similix divides all primitive operators into two classes: *transparent* and *non-transparent*. Applications of transparent operators to all static arguments are annotated static by BTA and are reduced during partial evaluation by applying the function to the set of known, static arguments. Applications of non-transparent operators are always annotated dynamic and are never reduced. In general all operators that can be correctly reduced during partial evaluation are considered transparent. The exceptions are side-effecting operators and I/O operators. However, the Similix user can modify the default classifications of operators as transparent or non-transparent.

Similix 5.0 further subdivides the transparent operators into two groups: transparent and *transparent-if-needed*. Transparent-if-needed are operators only reduced if the reductions are necessary to compute the value of the predicate of a conditional. Otherwise, transparent-if-need operators function just like non-transparent operators.

Transparent operators are reclassified as transparent-if-needed if their output can increase in the size of the input. For example, `cons`, `list`, `append`, `+`, `-`, `*`, and `/` are all transparent-if-needed; whereas, `cdr`, ..., `cddddr`, `pair?`, `null?`, `list?`, `equal?`, `reverse`, `list-ref`, and `member` are transparent. As for the original division between transparent and non-transparent operators, the Similix 5.0 user can override the defaults as to which operators are transparent and which are transparent-if-needed.

The addition of is-used analysis to Similix 5.0 enables it to terminate when performing partial evaluation of the iota function in Figure 4.7 on page 141, repeated as Figure 4.20, applied to an unknown input. Is-used analysis annotates the variable `i` as not being used. As a result, the BTA annotation of `i` is converted from static

```
(define iota
  (lambda (n)
    (loop 1 n)))

(define loop
  (lambda (i n)
    (if (= i n)
        '()
        (cons
         i
         (loop (1+ i) n)))))
```

Figure 4.20: First-order iota function (repeat of Figure 4.7)

to dynamic in the dummy function application inserted for the dynamic conditional. Since both arguments of the inserted application are dynamic, only one specialization is created. Partial evaluation terminates after creating the single specialization.

Limitations and Advantages of Similix 5.0

Similix 5.0 overcomes most of the limitations of the termination mechanism of Mix. It handles a source language including higher-order functions, utilizes a BTA supporting partially static structures, and due to is-used analysis terminates when processing `iota` applied to an unknown input value. However, Similix continues to suffer the limitations of a monovariant BTA. Furthermore, Similx 5.0 fails to terminate in some cases in which lazy use analysis yields termination.

Is-used analysis is limited in two regards. It lacks the concept of a value being *partially used* in a sense akin to a partially static value being partially known. And, it records values as being used even if they contribute to making control flow decisions that do not effect the result of a program. Each of these limitations is addressed in order.

Partial Use: Partially static BTA was invented in order to produce better results when analyzing programs containing values partially known and partially unknown during partial evaluation. Similarly, partial use is needed in order to produce better

```
(define iota
  (lambda(n cons-or-vector)
    (loop (cons 1 cons-or-vector) n)))

(define loop
  (lambda (i-and-cons-or-vector n)
    (if (= (car i-and-cons-or-vector) n)
        '()
        (if (equal? (cdr i-and-cons-or-vector) 'cons)
            (cons
             (car i-and-cons-or-vector)
             (loop (cons (+ 1 (car i-and-cons-or-vector))
                         (cdr i-and-cons-or-vector))
                   n))
            (vector
             (car i-and-cons-or-vector)
             (loop (cons (+ 1 (car i-and-cons-or-vector))
                         (cdr i-and-cons-or-vector))
                   n))))))
```

Figure 4.21: An unusual version of iota that returns the result composed of either cons cells or vectors and passes two arguments as a pair

results when is-used analysis is applied to a program with values that are only partially used in making control flow decisions. Take for example the unusual version of iota in Figure 4.21. This version of iota returns a result either composed of cons cells or vectors. `Loop` passes two conceptual arguments as a pair, the index of the iteration and a flag telling whether to use cons cells or vectors. Imagine partial evaluation of this version of iota applied to an unknown value and `'cons`. Because half of the argument to `loop`, the type of structure utilized to build the result, is used in evaluating the predicate of the inner conditional, the entire variable `i-and-cons-or-vector` is annotated as being used by is-used analysis. Consequently, `i-and-cons-or-vector` remains static even after is-used analysis. This causes partial evaluation to diverge since each iteration of the `loop` recursion produces a new value for the `i` portion of `i-and-cons-or-vector`.

The version of iota in Figure 4.21 presents a problem for is-used analysis because two portions of a value that serve different purposes have been combined into a single object. A version of is-used analysis supporting partial use would annotate the variable `i-and-cons-or-vector` as being partially used. The car of the cons cell would be annotated as unused and the cdr as used. As a result, the car of `i-and-cons-or-vector` would be reannotated as dynamic by is-used analysis. This would result in convergence of partial evaluation. Lazy use analysis inherently captures this form of partial use.

What Does It Mean To Be Used?: The second limitation of is-used analysis is illuminated by considering the evolution of use analysis. I first developed what was later to be called eager use analysis. Eager use analysis often diverged due to use of information in performing delta reductions as part of computations that ought to have been dead code eliminated, in whole or part. The next step in the evolution was the design of an analysis that functioned very similar to lazy use analysis, but asserted use of information any time it was utilized in making a control flow decision. This analysis similarly was found to diverge due to information being used in making control flow decision that ought to have been dead code eliminated. The final step in the process was the creation of lazy use analysis.

Is-used analysis might aptly be described as a static version of the intermediate use analysis described above.[5] It shares with that analysis the inability to enforce termination for the strange version of the iota function shown in figure 4.22. Consider partial evaluation of `iota` applied to an unknown input. The entire inner conditional can be dead code eliminated. However, is used analysis annotates `j` as used because of the predicate of the inner conditional. As a result, the inserted specialization point for the outer conditional retains `j` as a static parameter. As the value of `j` changes for each recursive call to `loop`, an infinite number of specializations are created.

Advantages of Similix 5.0

The major advantage of Similix's termination mechanism is its computational efficiency. All three analyses, flow analysis, BTA, and is-used analysis, have been demonstrated to have good complexity bounds and efficient run times. Furthermore,

[5]These two analyses were developed during the same time period and independently. Eager use analysis was published prior to the design of either of these analyses and is referenced in publications presenting is-used analysis.

```
(define loop
  (lambda (i n j)
    (if (= i n)
        ((lambda (a b) a)
         '()
         (if (> j 3)
             '()
             #t))
        (cons
         i
         (loop (+ 1 i) n (+ 1 j))))))
```

Figure 4.22: An unusual version of iota ends the result with either the empty list or `#t`

with the addition of is-used analysis, Similix terminates on a sufficiently broad class of programs to make it a reasonably useful tool, as opposed to just a research curiosity. Similix may be unique in this regard.

4.1.5 Fuse

Fuse is an online partial evaluator. Its termination mechanism, as described in [48], is very simple and fairly aggressive. While Fuse terminates on all input programs not containing infinite loops, it is susceptible to premature termination.[6]

Fuse's termination mechanism is based on two primary data structures, a cache of specializations created for each source function and a stack of pending applications. The cache is indexed by the source function and the arguments for which the associated specializations are created. The stack contains a description of each pending application as well as a marker designating each point in the execution sequence at which a dynamic conditional was encountered.

When symbolic execution of an application is about to be performed in Fuse, the cache is first checked to see if a specialization already exists for the given function and argument values. If one does, then a residual application is created that applies the specialization found in the cache, and symbolic execution of the current thread

[6]Later versions of Fuse included other approaches to termination including manual finiteness annotations, capturing the creation-time stack in closures [37], and detection of static sized structures.

of execution is completed. If there is not a hit in the cache, the stack is searched in order to determine whether a recursive call is about to be performed. The search starts with the most recent application and proceeds backwards in time.

If the pending application is not determined to be a recursive call, symbolic execution of the application is performed. If the application is a recursive call and the recursion does not span a dynamic conditional, symbolic execution of the pending application is also performed. A recursion not spanning a dynamic conditional can only diverge if the source program would diverge at runtime if it reached the entry to the recursion. However, if a recursive loop spans a dynamic conditional, Fuse terminates the recursion.

Once Fuse terminates a recursion, it needs to find a specialization for the recursive loop. Fuse starts by taking the corresponding arguments of the initial application and the recursive application and generalizing them to form a new argument set. The cache is then checked to see if a specialization already exists for an application of the pending function to the generalized argument set. If one does, then a residual call to the specialization in the cache is created. Otherwise, symbolic execution of the application of the the given function to the generalized argument set is performed in order to create the needed specialization.

Limitations and Advantages of of Fuse

The major limitation of Fuse's termination mechanism is premature termination. Fuse can produce suboptimal residual code due to the failure to create some desirable specializations. It can also fail to produce the most highly optimized code for certain specializations.

Consider once again the regular expression matcher from Figures 4.4 and 4.5 on pages 139 and 140, repeated in part in Figure 4.23. Partial evaluation of `match?` applied to the pattern kleene star of `a` and an unknown input string initiates symbolic execution of `match?`. This leads to an application of `match-pattern?` to the kleene star pattern, a null pattern, and an unspecified input. The `cond` in `match-pattern?` executes the kleene star case applying `match-star?` to the identical set of inputs. Since the input is unknown, the conditional in `match-star?` is dynamic. When the second branch of the conditional is executed, `match-pattern?` is applied to the term `a`, the original kleene star pattern, and an unknown input. Fuse detects this recursive application of `match-pattern?` spanning a dynamic conditional and terminates. Generalization of the corresponding arguments of the two applications results in creation of a specialization of `match-pattern?` for two unknown patterns and an unknown input string. The specialization created and the residual program produced are largely unoptimized due to the unknown nature of the generalized arguments.

Fuse's termination mechanism is limited by its assumption that all recursive applications are equivalent. It has no way of differentiating amongst different recursive applications of the same function. It is unable to decide, for example, that the first recursive application is distinct from the original application, but the second recursive application is equivalent to the original application. This difference between Fuse's termination mechanism and use analysis leads Fuse to terminate prematurely on the regular expression matcher, whereas use analysis proceeds until a sufficient number of iterations of the recursion are performed in order to produce better residual code.

The major advantage of Fuse's termination mechanism is that by aggressively terminating, it terminates on some programs on which use analysis does not. Neither Fuse nor use analysis, however, guarantees termination for all input programs, unlike finiteness analysis.

4.1.6 Conclusion

Lazy use analysis has been compared with the significant alternative termination mechanisms of Mix, Finiteness Analysis, Similix, and Fuse. Each of the alternatives has been shown to fail to handle appropriately one or more of the types of source code motivating the development of use analysis. However, each of the alternatives is significantly more efficient than use analysis. The basis of use analysis' efficiency problems and some potential solutions to them are addressed in Chapter 5.

```
(define match?
  (lambda (pattern input) (match-pattern? pattern null-pattern input)))

(define match-pattern?
  (lambda (pattern rest-pattern input)
    (cond ((null-pattern? pattern)
           (match-null? rest-pattern input))
          ((term? pattern)
           (match-term? pattern rest-pattern input))
          ((kleene-star? pattern)
           (match-star? pattern rest-pattern input))
          ((concat? pattern)
           (match-concat? pattern rest-pattern input)))))

(define match-term?
  (lambda (term-pattern rest-pattern input)
    (if (and (pair? input)
             (equal? (term-symbol term-pattern) (car input)))
        (match? rest-pattern (cdr input))
        #f)))

(define match-star?
  (lambda (star-pattern rest-pattern input)
    (or (match? rest-pattern input)
        (match-pattern? (kleene-star-expr star-pattern)
                        (concat star-pattern rest-pattern)
                        input))))

(define match-concat?
  (lambda (concat-pattern rest-pattern input)
    (match-pattern? (concat-head concat-pattern)
                    (concat (concat-tail concat-pattern) rest-pattern)
                    input)))

(define match-null?
  (lambda (rest-pattern input)
    (if (null-pattern? rest-pattern)
        (null? input)
        (match? rest-pattern input))))
```

Figure 4.23: A program for matching regular expressions (repeat of Figure 4.4)

4.2 CPS Conversion

Fisher and Plotkin's continuation-passing-style (CPS) conversion [19, 33] has played a central role in the compilation of Scheme starting with Steele's first Scheme compiler [44]. Danvy advocates CPS conversion as a stage in offline partial evaluation in [14]. He observes that abstract interpretation after CPS conversion yields more precise results, improving the precision of BTA and therefore the quality of residual code produced.

Ruf identifies several sources of loss of information causing decreased residual code quality in most partial evaluators in [37]. These include failure to represent the return values of residual applications of specializations as anything but completely unknown values, generalization of the results of both branches of a conditional, and generalization of argument values from multiple iterations of a recursion in order to build a specialization. CPS conversion participates in the amelioration or elimination of each of these sources of loss of information.

All three of the above sources of information loss result from coalescing and approximating values created along multiple flows of control into a single value correct for all the flows of control. The unknown value is clearly a correct approximation for all return values from a specialization. Analyzing the continuation of a conditional using a generalization of the results of the two flows through a conditional, one for the consequent and the other for the conditional, is a correctness preserving approximation to analyzing the continuation of the conditional separately using the result of each flow of control through the conditional. Finally, a generalization of the argument values for different iterations of a recursion for the purpose of creating a recursive specialization is an approximation to the effect of analyzing every possible iteration of the recursion as a separate flow of control.

How CPS conversion lessons the effect of each of the three sources of information loss will be discussed separately. This is followed by a presentation of the costs of CPS conversion. Finally, the role of CPS conversion in lazy use analysis, its benefits, and its costs are outlined.

4.2.1 Benefits of CPS Conversion

CPS conversion effectively eliminates returning values from functions. Consequently, approximation of return values of specializations is no longer an issue. However, the values previously returned by functions become arguments passed to continuations. In some cases, there may be losses of information in approximating the new arguments introduced by CPS conversion, so the loss of information previously taking place is not necessarily eliminated.

When a specialization is recursive, the approximation of the return value of the specialization is replaced by an approximation of an argument to a continuation. If the approximation algorithm for arguments for recursive specializations is more accurate than utilization of the completely unknown value, which it is for many partial evaluators, then the loss of information during partial evaluation of the CPS converted program is often less than for the original source program. This improvement in precision comes in addition to those resulting from the increased precision of BTA, for offline systems, addressed by Danvy.

CPS conversion does eliminate the loss of information about return values exhibited by most partial evaluators when applying non-recursive specializations. There is no approximation of the argument to the continuation replacing the return value of the specialization, so the previous complete lack of information regarding the return value is replaced by more complete information about the value passed to the continuation. Again, this improvement in precision comes in addition to any other improvements resulting from BTA being performed on a CPS converted source program.

CPS conversion eliminates the loss of information resulting from approximating the results of separate flows of control through a conditional by a single value. Each branch of a conditional applies the continuation of the conditional to the result of that branch. Separate analysis of the continuation of the conditional is performed for each flow of control through the conditional. This improvement in accuracy comes naturally as a result of CPS conversion. Special effort is required in order to combine the two flows of control after CPS conversion.

Separate analysis of the continuation of a conditional for each flow of control through the conditional could be achieved without CPS conversion. However, most partial evaluators consider the exit from a conditional to be a join point for the analysis, much as is the case for most traditional compiler analyses. They do not choose to analyze separately the continuation of the conditional for each flow of control into the continuation.

A more precise approximation of the argument values for creating specializations is a second order effect of CPS conversion directly resulting from the previously described benefits. More precise approximations of the return value of specializations contribute to computing more precise approximations of all computations involving those return values. The more precise results of these computations are often involved in the computation of argument values that are consequently also more precise. Similarly, the separating out of the analysis of different flows of control often leads to different recursions involving the same function no longer being conflated in the analysis. By separating different function applications into separate groups, the argument approximations for each group only need be correct for the applications belonging to that particular group. The approximations utilized for a single group often may be more precise than those resulting from considering all the applications in all of the groups as a whole.

4.2.2 Costs of CPS Conversion

The analysis benefits of CPS conversion come at the expense of increased computational cost to perform the analysis. Utilization of CPS conversion to separate out analysis of different flows of control yields more precise results because more analysis is being performed. Each join point in an analysis prior to CPS conversion eliminated as a result of CPS conversion doubles the amount of analysis performed of the continuation representing the rest of a computation starting at the previous join point. The total amount of increased computation is therefore exponential in the number of join points removed. While some of the theoretical increase in complexity may be eliminated in practice by detecting pieces of the analysis that are identical and therefore need not be replicated, in the worst case the increase in complexity is still exponential.

4.2.3 CPS and Lazy Use Analysis

CPS conversion is used as a prepass in a partial evaluator whose termination mechanism is based on lazy use analysis. Neither the exits from conditionals nor those from function applications are join points for lazy use analysis. The only conceptual[7] join points for lazy use analysis are those resulting from termination of recursions.

Lazy use analysis benefits from all of the advantages of increased precision of analysis resulting from CPS conversion described above. In addition, increased precision of the analysis aids lazy use analysis in making termination decisions. More precise uses can be computed for the more precise values generated during symbolic execution. These in turn yield improved termination, both in terms of detecting equivalent iterations and those that are distinct.

Not surprisingly, the first implementation of a partial evaluator based on lazy use analysis suffers from performance problems resulting from the exponential number of flows of control that are considered by the analysis. This and related problems are discussed in more detail in Chapter 5.

[7]The implementation does join equivalent pieces of the analysis in order to increase efficiency as described in Chapter 5.

4.3 Two Phase Partial Evaluation[8]

Both Osgood [32] and I independently discovered the idea and benefits of what I call two phase partial evaluation. Two phase partial evaluators separate the analysis phase in which potential specializations are analyzed from a later code generation phase in which residual code is produced utilizing the potential specializations characterized by the analysis phase. This partitioning of partial evaluation enables analysis decisions, such as termination, to be separate from code generation decisions, such as what potential specialization to include in the residual code, what potential specialization to apply at each call site, and when to inline a specialization.

Two phase partial evaluation is different than the partitionings utilized by offline partial evaluators. The separation of BTA into a separate phase by offline partial evaluation is really a partitioning of the analysis into two passes, as opposed to a separation of analysis from code generation. BTA serves to enhance the execution time of partial evaluation; however, it does this at the expense of a decrease in the quality of residual code in some cases. This is significantly different from the partitioning in two phase partial evaluation designed to increase both the quality of residual code and the termination properties of partial evaluation, but not concerned with execution efficiency of partial evaluation.

Postunfolding, a phase present in many offline partial evaluators, is also somewhat different than the code generation phase of two phase partial evaluation. Most code generation decisions are made in offline partial evaluators prior to post unfolding. Postunfolding might best be thought of as a peephole optimization run after the completion of most of the code generation and designed to improve the residual code quality. By the time the postunfolding phase is run, it is too late to make most of the code generation decisions outline above that are performed during the code generation phase of a two phase partial evaluator.

Osgood observes that single phase partial evaluators tend to use *greedy* code generation algorithms. These might aptly be compared to hill climbing algorithms that perform local optimization at each point in their execution, possibly resulting in local,

[8]The second phase of my partial evaluator was designed, but never implemented.

but not global, optimization. Osgood further states these approaches often lead to residual code explosion and resulting performance degradation. Attempts to eliminate this problem often lead instead to an inappropriate or insufficient selection of specializations.

Among the optimizations performed by PARTICLE, Osgood's partial evaluator, are the following types of optimizations that cannot be performed at all, or as effectively, by a one phase system: *call site retargeting*, *multispecialization optimization*, and selecting specializations based on accurate global cost/benefit information. Call site retargeting is the decision to call a specialization that is correct for more general arguments than those supplied at a given call site in order to obviate the need to create residual code for the more specific specialization. This optimization chooses to trade off the runtime efficiency of the more specific specialization against the costs of increased residual code. Successful implementation of this optimization is dependent on a knowledge of the execution costs of the alternative specializations, the size of the residual code for each specialization, and the relative frequency with which the specializations will be called at runtime. None of these values tend to be available on the fly during one phase partial evaluation.

Multispecialization optimization is a global technique in which decisions regarding what specializations to include in a residual program and which to apply at each call site are not made independently, but collectively based on a global analysis of the costs and benefits. By definition, this type of global optimization cannot be performed on the fly since all the necessary information is not available until analysis has been performed of all possible specializations to be utilized in a residual program.

Finally, even simple decisions like whether to inline specializations cannot be made without global cost/benefit information. For example, it nearly always improves performance to inline a specialization that is only applied at a single call site since residual code size is reduced as well as the runtime cost of applying the specialization.[9] Again, any optimizations requiring global information cannot be performed by one phase systems.

[9]The only exception is in rare cases in which the inlining causes pernicious cache effects on the target machine.

To date no implementation of the code generation phase exists for any partial evaluator whose termination mechanism is based on lazy use analysis. Consequently, it is impossible to make any comparative claims to those of Osgood regarding two phase partial evaluation. What can be observed is that partitioning of partial evaluation into two phases enables all of the code generation optimization techniques developed by both the partial evaluation and traditional compiler communities to be brought to bear on the partial evaluation code generation problem, unfettered by considerations of the analysis phase such as termination.

Chapter 5

Large Resource Consumption

Lazy use analysis as currently implemented suffers from large resource consumption. Most serious is the huge amount of memory consumed when partially evaluating even reasonably small source programs. For example, partial evaluation of the regular expression matcher in Figures 4.4 and 4.5 on pages 139 and 140 applied to the pattern a^* and an unknown input string cannot be completed on any machine to which I have access. Based on an evaluation of the partially completed analysis, a machine with roughly 12-16 Gbytes of swap space would be required to complete the analysis utilizing the present implementation. During the analysis tens of thousands of applications and throws would be analyzed, creating tens of thousands of application records, as well as hundreds of thousands of symbolic values and use dependences.

This chapter begins by describing the most important sources of large resource consumption by lazy use analysis. Both those sources inherent in the lazy use analysis algorithm and those resulting from the implementation strategy selected are discussed. This is followed by a presentation of the approaches taken so far to decrease the resource consumption of the current implementation. The discussion concludes with an outline of some untested ideas potentially yielding a further reduction in resource consumption. A table including all sources of large resource consumption discussed appears in Figure 5.1. It specifies whether the problem is inherent to the use analysis algorithm or results from an implementation deficiency, any ideas for reducing the resource consumption, and whether the ideas have already been implemented and tested or not.

Source	Inherent/ Implementation	Solution	Implemented/ Untested
Separate analysis of distinct flows of control	Inherent	Join identical threads	Implemented
		Reuse of analysis	Untested
		Static analysis	Untested
Expanded argument sets	Inherent	Avoiding copying	Implemented
		Static analysis	Untested
Computing generlized return values	Inherent		
Retention of intermediate data structures	Inherent		
Non-generational stop & copy GC	Implementation	Different GC	Untested
Inefficient representations	Implementation	Change representations	Untested
		Reimplement in another language	Untested

Figure 5.1: Table of sources of large resource consumption

5.1 Sources of Large Resource Consumption

The sources of large resource consumption by lazy use analysis can be partitioned into two categories: those resulting from inherent aspects of the algorithm and those resulting from deficiencies in the implementation. This discussion begins by presenting the former and then proceeds to the later.

5.1.1 Inherent Costs of Lazy Use Analysis

There are four primary inherent sources of large computational resource consumption by lazy use analysis. All result from the attempt to perform more precise analysis than other partial evaluators in order to yield a better combination of termination and residual code quality. The first source of large resource consumption is the very aggressive approach taken to separate analysis of distinct flows of control. The second is expanding the effective argument set of every function and application to include not only the actual arguments, but also all the lexically apparent values referenced from the body of the function. The third is computing generalized return values for terminated recursions. And, the fourth is the need to retain most intermediate data structures created during the analysis until the entire analysis has been completed. Each of these sources of resource consumption is discussed in order.

Separate Analysis of Distinct Flows of Control

One of the limitations on the precision of the analysis performed by any partial evaluator is the selection of the join points in the analysis. A join is performed any time the results of two separate flows of control are combined and subsequent analysis is performed using the combined result, instead of being performed separately for the results of each flow of control. The types of join points utilized by different analyses can be broken into a number of categories: applications, exits from conditionals, and exits from applications, amongst others. Each of these cases is discussed in principal, followed by a presentation of in what cases lazy use analysis performs a join for each category.

Every function application is a join point for monovariant partial evaluators. Since only a single specialization is created for each source function, all analysis of applications of each function is combined. Polyvariant partial evaluators don't join analysis of all applications of each function together, but selectively join applications based on the termination mechanism utilized.

Exits from conditionals are join points for most partial evaluators. The continuation of the conditional is analyzed based on the combined results of the analysis of the consequent and the alternative, rather than being analyzed separately based on which one of the consequent and alternative is passing control to the continuation. When an N-way conditional appears in a function, N different flows of control result.

Since lazy use analysis analyzes at least two iterations of all recursions before making a termination decision, at least N^2 flows of control are analyzed for a recursive the function containing an N-way branch. Computation of a generalized return value entails analysis of at least another two iterations of the recursion. The net result is analysis of at least N^4 flows of control. This represents significantly more analysis, and therefore greater resource consumption, than is utilized by an algorithm for which exits from conditionals are join points.

Most partial evaluators only analyze the continuation of a function application once for each application. Some analyze the continuation separately for each flow of control within the function returning a distinct result. Much as for conditionals, N different return values implies N times the analysis of the continuation when function returns are not join points. When multiple conditionals are considered and/or recursions are analyzed more than two iterations deep, the number of flows of control to be analyzed can grow exponentially in the number of branch points and the depth of recursion analyzed.

Lazy use analysis as implemented strives to maximize the precision of the analysis by taking a very aggressive approach to performing separate analysis for different flows of control. Joins are only performed when the analysis believes them to be required in order to facilitate termination. Exits from dynamic conditionals are never join points. The continuations of dynamic conditionals are always separately analyzed based on whether control is passed to the continuation by the consequent or the alternative. Function exits are also never join points. The continuations of function applications are separately analyzed for each flow of control within a function producing a return value from the application. Function applications are only join points when lazy use analysis' termination mechanism decides a recursion needs to be terminated. In this case, the join is between the two applications deemed equivalent.

A polyvariant partial evaluator utilizing exits from conditionals and function applications as join points effectively loses these join points if its source programs are CPS converted prior to partial evaluation. This is because the exits become new function applications implementing the throws to the continuations. So long as the polyvariant specializer does not perform a join at the function application points, the joins for the exits effectively disappear. This explains why performing CPS conversion prior to many analyses improves the precision of those analyses.

Lazy use analysis performs CPS conversion during preprocessing of the source program in order to achieve the increased precision yielded by eliminating exits from conditionals and functions as join points. The downside of this approach is the significant increase in the number of total expressions processed by the analysis. In the worst case, the time complexity of the analysis increases exponentially in the number of potential join points removed. Since the memory consumption of lazy use analysis is also roughly linear in the number of expressions analyzed, this explains one source of very large memory consumption by lazy use analysis in comparison with analyses utilizing exits as join points.

Most partial evaluators consider any two applications of the same function to equivalent arguments to be equivalent. Some partial evaluators only consider a subset of the arguments or a subset of the information present in each argument when making equivalence decisions regarding applications. In all of these partial evaluators, the equivalent applications serve as join points of the analysis. Since lazy use analysis is based on the CS-DOS, two applications with identical arguments are not necessarily deemed equivalent unless their continuations utilize their return values in an equivalent fashion. This means identical function calls are separately analyzed by use analysis when they have nonidentical contexts. This leads to a considerable amount of additional analysis for many programs.

For example, partial evaluation of the regular expression matcher in Figures 4.4 and 4.5 on pages 139 and 140, repeated in part in Figure 5.2, for the pattern a^* and an unknown input eventually leads to symbolic execution of `match-star?` applied to a^*, the null pattern, and the unknown input. Continued symbolic execution of `match-pattern?` in the body of `match-star?` eventually leads to an identical application of `match-star?` applied to a^*, the null pattern, and the unknown input. Most partial evaluators would perform a join at this point. However, lazy use analysis does not perform a join because the two applications of `match-star?` have non-identical continuations.

Expanded Argument Sets

Correct termination of higher-order programs requires considering not only the arguments to functions, but also all lexically apparent values utilized in the body of a function. Lazy use analysis performs an implicit lambda lifting [27], removing the lexical hierarchy and making all passing of information explicit, in order to address this need. This yields between a two and five times increase in the arity of most functions. The increased arity adds a significant constant factor to the memory required to store the use dependence graph and represent function applications. It also increases the computational cost of processing every application, propagating use information, etc.

```
(define match?
  (lambda (pattern input) (match-pattern? pattern null-pattern input)))

(define match-pattern?
  (lambda (pattern rest-pattern input)
    (cond ((null-pattern? pattern)
           (match-null? rest-pattern input))
          ((term? pattern)
           (match-term? pattern rest-pattern input))
          ((kleene-star? pattern)
           (match-star? pattern rest-pattern input))
          ((concat? pattern)
           (match-concat? pattern rest-pattern input)))))

(define match-term?
  (lambda (term-pattern rest-pattern input)
    (if (and (pair? input)
             (equal? (term-symbol term-pattern) (car input)))
        (match? rest-pattern (cdr input))
        #f)))

(define match-star?
  (lambda (star-pattern rest-pattern input)
    (or (match? rest-pattern input)
        (match-pattern? (kleene-star-expr star-pattern)
                        (concat star-pattern rest-pattern)
                        input))))

(define match-concat?
  (lambda (concat-pattern rest-pattern input)
    (match-pattern? (concat-head concat-pattern)
                    (concat (concat-tail concat-pattern) rest-pattern)
                    input)))

(define match-null?
  (lambda (rest-pattern input)
    (if (null-pattern? rest-pattern)
        (null? input)
        (match? rest-pattern input))))
```

Figure 5.2: A program for matching regular expressions (repeat of Figure 4.4)

```
(define (fact n)
  (if (zero? n)
      1
      (mult 1 n)))

(define (mult i j)
  (if (= i j)
      i
      (let ((mid (truncate (/ (+ i j) 2))))
        (* (mult i mid)
           (mult (1+ mid) j)))))
```

Figure 5.3: Divide and Conquer Factorial

Computing Generalized Return Values

Lazy use analysis computes generalized return values in order to improve the precision of the analysis of the continuation of terminated recursions. The decision to use the lazy use analysis framework in order to compute a generalized return value tailored to the context in which a terminated recursion is applied increases the computational complexity of the analysis and therefore the amount of resources utilized. Ruf computed generalized return values independent of the context in which a recursive function is applied [37]. Consequently, he only needed to analyze each distinct generalized application at most once. My system must reanalyze each generalized application for each context in which a recursive function is applied.

For example, consider partial evaluation of the divide and conquer version of factorial in Figure 5.3 applied to an unknown argument value. Because there are two recursive calls to `mult`, in its body, symbolic execution of recursive calls to `mult` will be terminated several different times. For each terminated recursion, a generalized return value is required. Many of the terminated recursions are of generalized applications of `mult` applied to two unknown values. Despite each of the generalized applications having identical arguments, analysis to compute a generalized return value is performed multiple different times since the continuations are not identical in all cases.

As stated earlier, computation of generalized return values is performed by symbolic execution of generalized applications. Since lazy use analysis is utilized to terminate the computation of the generalized return value, all the factors outlined in the previous section leading to large resource consumption apply to these calculations as well. In fact, since generalized applications utilize less specific arguments, they tend to lead to a greater number of flows of control. The result is often even greater resource consumption to compute generalized return values than to perform the rest of the analysis.

The reasons for huge resource costs become clear when one realizes the different sources of resource consumption tend to combine multiplicatively, not additively. Computing generalized return values utilizing lazy use analysis means the analysis must symbolically execute at least four iterations of any terminated recursion. At least two iterations are required before an initial termination decision is possible. At least two additional iteration are required to terminate the recursion initiated by symbolic execution of the generalized application. And, the two additional iterations for computing a generalized return value are performed for each flow of control in which a recursive call is made.

Consider partial evaluation of the regular expression matcher in Figure 5.2 applied to an unknown regular expression and pattern. Each application of `match-pattern?` generates four flows of control for the four branches of the `cond`. Each of these flows of control symbolically executes four iterations of the main recursion in order both to terminate the initial recursion and the computation of the generalized return values. The net result is analysis of 256 (i.e., 4^4) flows of control.

Retention of Intermediate Data Structures

Symbolic execution performed by lazy use analysis is a top down processes beginning with an initial function application initiating execution of a program and proceeding through return of one or more results. Assignment of use annotations to values utilized by symbolic execution and the termination mechanism is a bottom up process, starting with the return values and working backwards towards the initial inputs to a program. Since lazy use analysis switches back and forth between symbolic execution and propagation of use information as the analysis is performed, the use dependence graph, use annotations, symbolic values, and symbolic execution state information all must be retained until the analysis is completed.

State information about symbolic execution is necessary so symbolic execution of any terminated recursion can be reinitiated, if required. This also necessitates retention of symbolic values needed to reinitiate symbolic execution. The use dependence graph and use annotations are required so any changes in use annotations can be propagated to all appropriate values. Addition of new nodes to the use dependence graph requires retention of those nodes to which the new nodes are to be attached and retention of the use annotations on those nodes so those annotations can be propagated over the new dependences.

5.1.2 Unnecessary Resource Consumption

There are two main sources of unnecessary memory consumption by my current implementation of lazy use analysis. The Scheme system being used to execute my partial evaluator utilizes a non-generational stop-and-copy garbage collector. This form of garbage collection doubles the virtual address space required to execute a program, as it requires two equal sized heaps, only one of which is being utilized to store live data objects at any point in time.

The representations of symbolic values, use annotations, and use dependences were all selected to maximize flexibility, debugability, and efficiency of execution. Significantly more concise representations are certainly possible, but in some cases at the cost of decreased speed of symbolic execution. It appears the memory consumption of the data structures could be reduced by at least a factor of four. Combining this with the factor of two lost due to the form of garbage collection utilized, the memory image of lazy use analysis ought to be reducible by a factor of eight without significant conceptual effort. With somewhat more effort a port of the algorithm to a language not requiring 8 byte words (on a 64-bit architecture) to represent the smallest denotable values would significantly reduce the memory requirements.

5.1.3 Steps Taken to Reduce Memory Consumption

Two major approaches have been taken to reduce the memory consumption of lazy use analysis: joining identical computations to avoid redundant analysis and representation and elimination of copying symbolic values when the two copies are guaranteed to have identical use annotations. Each of these optimizations is explained below, including an analysis of the effectiveness of each optimization.

Avoiding Redundant Analysis

Any time two computations are identical, they need not be analyzed separately. Most partial evaluators keep a cache of specializations in order to avoid redundant analysis. When symbolic execution reaches an application, the function and arguments are compared with all the specializations in the cache in order to determine whether an identical application has already been analyzed. If it has, the results of the previous analysis are reused.

As explained previously, the definition of an identical application is less general for lazy use analysis than for algorithms based on approximating the CF-DOS. Not only must functions be equivalent and all the actual and inherited arguments be identical, but also the continuation must be identical. This significantly limits the opportunities to perform this optimization. In practice, it is rarely, if ever, applicable.

A related optimization is the detection of identical throws. These are cases in which a throw is performed to the same continuation utilizing an identical argument and identical inherited values. Identical throws occur slightly more frequently than identical applications, but still not that often. In most cases they result at locations in the analysis that would be join points for other algorithms. For example, at exits from conditionals when both flows of control through the conditional return identical values or at exists from functions when more than one flow of control through the function returns an identical value.

Avoiding Creating Identical Symbolic Values

The use analysis algorithm specifies that each time a function application is performed, each argument is copied, an identity use dependence is created between the originals and the copies, and symbolic execution of the function body is performed utilizing the copies. Copying takes place in order to ensure different uses of the same value by different functions are not conflated. However, conflating of uses can only occur when the same value is utilized in more than one computation.

The copy avoidance optimization is based on statically detecting during preprocessing variables only referenced in one location in the preprocessed source code. The values stored in such variables never need to be copied prior to being passed as an argument to a function since they by definition can only be utilized in a single computation. In order for a value to be used in more than one computation, some variable bound to the value at some point must be referenced in more than one location in the source code. The symbolic value is copied only when accessed through one of the multiple references to the same variable.

The copy avoidance optimization based on detecting single use variables has a significant effect on the number of symbolic values, use annotations, and use dependences created. It also significantly reduces the computational cost of propagating use information over the use dependence graph due to the reduced size of the graph. On an ensemble of programs tested, well over 80% of variable references were marked as single reference by the preprocessor and the creation of well over 80% of potential copies of symbolic values were avoided. In particular, most of the inherited arguments added to function definitions are annotated as single use, greatly reducing the cost of the increased arity. However, the representations of the applications continue to require slots in which to store every one of the arguments, whether copied or not. Therefore, copy avoidance does not completely do away with the memory costs of increased arity.

5.2 Untested Ideas to Reduce Resource Consumption

I have developed two additional untested idea for reducing the resource consumption of lazy use analysis. The first is an approach to reusing the results of analysis of similar applications in order to avoid creation of redundant data structures. The second is the use of static analysis prior to lazy use analysis in order to eliminate the need to perform some of the dynamic analysis. Each of these two possibilities is presented below.

5.2.1 Reusing Analysis

The objective of this optimization is to identify function applications symbolic execution of which will perform the identical reductions as those performed for symbolic execution of some other application. In this case, the two applications will produce identical use dependence graphs, so both execution time and memory utilization can be reduced by reusing the results of symbolic execution of one application.

Detection of applications producing equivalent symbolic execution is facilitated by an eager use analysis for reuse. As was explained in Chapter 2, eager use analysis records the information utilized in performing reductions during symbolic execution. Since performance of identical reductions implies production of identical use dependences, symbolic execution of two applications generating identical eager use characterizations must also create identical use dependence graphs.

For any use, there is some set of values *compliant* with that use. These are the values that can correctly supply the information reflected in the use. If the same domains of values are used for values and uses, then the values, c, compliant with a use, u, are those values such that $u \preceq c$, for the $\preceq$ operator defined in Chapter 2. For example, any integer and $\perp_{Int}$ are both compliant with the use $\perp_{Int}$. However, `#t` is not compliant with the use $\perp_{Int}$. Similarly, all values are compliant with the use $\perp_{PEval}$.

An eager or lazy use analysis overspecifying use when it must approximate characterizes the class of specializations that can correctly be used for any set of argument values compliant with the use annotations computed. What the analyses do not do is specify when separate analysis of a compliant function application might produce more detailed information about a new function application, yielding a better specialization. Use compliance ensures the information collected is correct for any use compliant application, but not that the description is the most precise one possible. For example, symbolic execution and eager use analysis of `(+` $\perp_{Int}$ $\perp_{Int}$`)` produces the result $\perp_{Int}$ and a use annotation of $\perp_{Int}$ for each argument. The function application `(+ 2 3)` is use compliant with the potential specialization generated for `(+` $\perp_{Int}$ $\perp_{Int}$`)` and the analysis for `(+` $\perp_{Int}$ $\perp_{Int}$`)` correctly characterizes the result of `(+ 2 3)` as an integer, but that characterization is not as precise as the one resulting from performing a separate analysis of `(+ 2 3)`. A separate analysis yields the result `5` and uses of the integer values of the arguments `2` and `3`.

A way of classifying those applications that are both use compliant and for which performing a separate analysis would not yield more detailed information is needed. *Demand analysis*, to be discussed in more detail below, is designed to achieve this end. Demand analysis is in some sense the dual of use analysis. Whereas use analysis represents the information utilized in performing symbolic execution of a computation, demand analysis represents the least amount of additional information necessary for symbolic execution to yield a more precise result.[1]

In general, performing eager use analysis of a function application requires performing symbolic execution of the application; so, it might not appear that eager use analysis helps reduce the amount of analysis and memory necessary. However, there are some special cases in which the results of eager use analysis can be determined without performing symbolic execution. Referring back to Figure 2.48 on page 73, repeated in Figure 5.4, eager use analysis always records utilization of some subset of the information present in the value of each argument in an application. Let's assume use analysis has already been performed for some application. Symbolic execution of a new application of the same function for which all of the argument values fall between the values in the original application and their corresponding use annotations, as represented by EU_{reuse}, generates identical EU_{reuse} annotations and an identical potential specialization, so long as the same termination decisions are made for all recursions.

The optimization I propose, which is the same as that in [37], is to compare the function and arguments of each application, before it is symbolically executed, with every application already analyzed. If the arguments of the new application all fall between the values of the arguments in the previous application and eager use

[1]Demand analysis was first implemented and utilized for optimizing reuse decisions by Ruf [37].

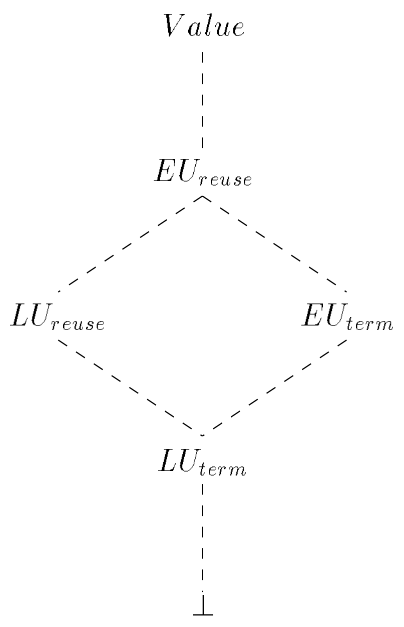

Figure 5.4: Relative amount of use information recorded by different analyses (repeat of Figure 2.48)

annotations of those arguments, then the new application need not be symbolically executed. The results of the previous analysis can be reused. The arguments of the new application and its continuation just need to be tied to the existing use dependence graph. In addition, the continuation of the new application must be analyzed for each of the results returned by the previously analyzed application.

There are three subtleties in reusing previous analysis. The first involves the way return values passed to the new continuation are created; the second, with how to handle the case when the new continuation utilizes the results in a different manner than the original continuation; and, the third with nonmonotonicity that can be introduced into the creation of generalized return values. Each of these subtleties is discussed in order. Finally, a proposal for increasing the cases in which this optimization is applicable by utilizing demand analysis is presented.

Generating Return Values When Reusing Analysis

Symbolic execution of multiple different applications of the same function can return very different results even though the eager use annotations and potential

```
(define id
  (lambda (val) val))
```

Figure 5.5: Identity Function

specializations produced by the analysis are identical for all of the applications. Eager use analysis only records information as being used when a computation is performed utilizing a value. Information may appear in a result, yet not be recorded as being used, if it is contained in a result value, but not used in any computation. For example, eager use analysis of the identity function in Figure 5.5 yields a use annotation of $\bot_{PEval}$ for `val`. The argument value is unused, but must be produced, since a later computation might utilize the result. Furthermore, the result of the identity function is very different for different arguments.

Efficient reuse of symbolic execution performed as part of eager analysis is achieved by parameterizing the analysis over those portions of the argument values unused during the analysis. The arguments to an application are annotated to encode from which application and argument position they originate. The portions of the return value(s) of a function originating from copies of pieces of arguments, rather than from some computation performed by the function, remain annotated in the return result(s). Correct return results for a different set of arguments can be created by copying the return results of the previous application, only with the correct portions of the new arguments substituted for the appropriate pieces if the old results.

For the identity function, achieving this result is simple. The single result is annotated as being the first argument of the application. When a new application of the identity function must be analyzed, all that is required is to substitute the argument of the new application as the result. However, when the body of a function utilizes many constructors and accessors, the process of substituting in the correct pieces of new arguments becomes somewhat more complex, but appears to be fairly straightforward and reasonably efficient. While multiple implementations are possible, the costs of this approach seem to include a small bookkeeping cost in implementing symbolic execution of each accessor and constructor, as well as the need to compute the pieces of the arguments to be substituted into the return results when generating new copies of the results.

Different Uses by Different Continuations

How to handle the case when the continuations of two applications for which a single symbolic execution of the function body has been performed utilize the results of the function applications differently presents two problems. On the pragmatic, implementation side there is the issue of how to percolate the different uses over a shared piece of a use dependence graph so the arguments for each of the different applications receive the appropriate use annotations. On the algorithmic side, there is the question of how to undo the sharing when the different uses result in the need to make different termination decisions for recursive applications in the body of a function. When different termination decisions are made, the two applications do not yield equivalent potential specializations so sharing of analysis is no longer possible. Each of these considerations is discussed in order.

The best approach for propagating different use annotations for different continuations over the same portion of a use dependence graph is not immediately obvious, but also does not appear to be a fundamentally difficult problem. One of many possible approaches is to dynamically create different memoization cells for lazy use annotations for each symbolic value any time different annotations are required for different flows of control sharing a piece of the use dependence graph. It appears at first blush this could be achieved both fairly simply and efficiently. Updating of the correct memoization cell could be ensured by tagging the use information as it travels over the use dependences based on the continuation from which it originates. While in theory, the tagging could become long and unwieldy, in practice I expect the tag lists would be short and easy to process. Finally, multiple memoization cells might be created for some symbolic values due to use information about different flows of control traveling through the use dependence graph sequentially and therefore not arriving at every node at the same times. To the extent nodes end up having multiple memoization cells containing identical annotations, these could be coalesced by the use propagation process and garbage collection could be used to reclaim the unneeded cells.

The splitting of shared analysis is slightly more complex problem. Use change daemons must be augmented so they handle the multiple memoization cells now possible for each symbolic value and detect changes in any of the memoization cells for a value. When activated, the daemons must not only detect whether symbolic execution of a recursion should be reinitiated, but also, whether a piece of shared analysis must be split as a result of changes in one memoization cell, but not another.

Actual splitting of analysis is fairly simple. An inefficient, but straightforward, approach is just to separate out the problematic application and perform symbolic execution of that application from scratch, generating new, unshared results. Slightly more efficient is to terminate the reanalysis when it throws to its continuation and graft the result onto the appropriate existing analysis of that continuation. Yet one step better is to just copy all the data structures created by previous analysis and attach the arguments and results of the problematic application to the new copy.

The precise details of implementing this optimization remain unknown. However, it appears both to offer the promise of a significant improvement in memory consumption and to be computationally feasible. On the other hand, it does require the addition of a substantial amount of new mechanism and complexity to the implementation of the analysis.

Nonmonotonicity in Creating Generalized Return Values

Termination of computation of generalized return values is dependent on a monotonicity property. The property is that application of a function to a set of arguments containing less information produces a result with the same amount or less information. This property is true for symbolic execution of all functions and special forms in Scheme as long as no analysis is reused.

Nonmonotonicity can result from reuse of the incomplete analysis of a potential specialization. During the computation of a generalized return value the amount of information in the return value progressively increases as the different possible flows of control and return values are analyzed. Until the analysis is completed, the representation of the generalized return value may contain more information than it eventually will. In fact, the intermediate value for the generalized return value of an application might contain less information than the return value for an applications of the same function to arguments containing less information. Therefore, if the intermediate value of the generalized return value is utilized in a computation due to reuse of analysis of a potential specialization before it has been completed, a violation of the monotonicity property can result.

Ruf in Section 4.4.2 of [37] discusses in greater detail how nonmonotonicity can result in divergence when computing a generalized return value. He also presents an approach to ensuring monotonicity even if analysis is reused before computation of a generalized return value has been completed. How best to address the problem of ensuring monotonicity when reusing analysis within lazy use analysis remains an open question at this time.

Increasing Applicability of This Optimization

So far it has been explained that analysis of an application may be reused when a new application of the same function is performed utilizing arguments containing an amount of information between that of the original arguments and the amount utilized during symbolic execution based on those arguments. It is also possible in some cases to reuse analysis when the amount of information supplied in arguments is a superset of that supplied in a previously analyzed application. Whereas use analysis presents a lower bound on the amount of information needing to be supplied for analysis to be reused, *demand analysis* yields an upper bound of the amount of additional information supplied before reanalysis is desired.

Demand analysis is in some sense the dual of use analysis. Whereas use analysis represents the information utilized in performing symbolic execution of a computation, demand analysis represents the least amount of additional information necessary for symbolic execution to yield a more detailed result. As with use analysis, it is simplest to begin by explaining eager demand analysis and then move on to the lazy version.

Eager Demand Analysis

The same domains of values in Figure 2.22 on page 31, repeated as Figure 5.6, utilized for values during symbolic execution and for use annotations can also be utilized for demand annotations. A number of expressions and their corresponding eager demand annotations appear in Figure 5.7. No computation is possible for `(car` $\bot_{PEval}$`)`. Any amount of additional information about the argument at the very least would allow the type check of the argument to be performed during symbolic execution. The demand annotation $\bot_{PEval}$ signifies that any value containing some information enables more computation to be performed by the analysis. For the expression `(integer?` $\bot_{Int}$`)`, there is no amount of additional information about the argument enabling additional computation to be performed. All use compliant applications are guaranteed to produce the identical analysis. In the expression `(1+` $\bot_{Int}$`)`, the actual integer value of the argument would be needed to perform additional computation. In the expression `(+` $\bot_{Int}\bot_{Int}$`)`, the integer values of both arguments would be needed to perform more computation. However, demand analysis must be conservative so demand annotations are created stating that availability of the integer value of either argument should result in separate analysis. Of course, if the integer value of only one argument is supplied, the resulting analysis produces the identical result. Finally, for the expression `(+` $\bot_{PEval}\bot_{Int}$`)`, there is no amount of additional information about the second argument enabling additional computation to be performed unless at least the type of the first argument is known.

Eager demand analysis can be implemented by adding an extra field to every symbolic value, much as for use analysis. Demand annotations are initialized to $\top$ when symbolic values are created signifying that initially there is no amount of additional information about the corresponding value that would allow additional computation to be performed. As delta reductions and control flow operations are performed, new demand annotations are generated by taking the greatest lower bound (GLB) of existing demand annotations and the demands made by an operation currently being analyzed. The greatest lower bound is utilized for demand because a demand represents the least amount of additional information needed to yield a more detailed result. Lower points in an information lattice represent lesser amounts of information. Demand is the dual of use in the sense that it uses the GLB as opposed to the LUB.

Int	$=$	$\texttt{0} + \pm\texttt{1} + \pm\texttt{2} + \cdots$	integers
		$\perp_{Int}$	unspecified integer
$Bool$	$=$	$\texttt{true} + \texttt{false}$	booleans
		$\perp_{Bool}$	unspecified boolean
Sym	$=$	$\texttt{'a} + \texttt{'b} + \cdots$	symbols
		$\perp_{Sym}$	unspecified symbol
Nil	$=$	$\texttt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\perp_{Pair} \equiv \perp_{PEval} \times \perp_{PEval}$	unspecified pair
$Closure$	$=$	$Lambda \times Env$	closure values
		$\perp_{Clos}$	unspecified closure
Env	$=$	$(Id \rightarrow PEval)^{\star}$	environments
$Kval$	$=$	$Int + Bool + Nil + Pair + Closure$	known values
$Bots$	$=$	$\perp_{Int} + \perp_{Bool} + \perp_{Pair} + \perp_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \perp_{PEval}$	partial evaluation values
		$\perp_{PEval}$	unspecified value

Figure 5.6: Value domains for partial evaluation of a pure subset of Scheme (repeat of Figure 2.22)

Expression	Argument Demand Profiles
(car $\perp_{PEval}$)	$\perp_{PEval}$
(integer? $\perp_{Int}$)	$\top$
(1+ $\perp_{Int}$)	$\perp_{Int}$
(+ $\perp_{Int}\perp_{Int}$)	$\perp_{Int}, \perp_{Int}$
(+ $\perp_{PEval}\perp_{Int}$)	$\perp_{PEval}, \top$

Figure 5.7: Eager demand annotations

Lazy Demand Analysis

Laziness is incorporated into demand analysis in the same manner it is for use. Each delta reduction or conditional control flow operation generates demand dependences reflecting how demands of results imply demands of the inputs. The result is a demand dependence graph. Demand annotations are generated by asserting demands of the result(s) of programs and propagating the demands through the demand dependence graph. What demands to assert about results depends on whether an underspecifying or overspecifying analysis is desired. Both alternatives are discussed below.

Figure 5.8 shows the relative amount of demand and use information recorded for $Value$ based on performing different types of demand and use analysis. The bottom half of the diagram is identical to the one in Figure 5.4 on page 193, which compares different types of use analyses. The top half of the diagram is a mirror image of the bottom half, with use analyses replaced by corresponding demand analyses and overspecifying analyses replaced by underspecifying analysis.

What it means to overspecify or underspecify demand is slightly less obvious than for use. I define overspecification of demand to mean a lower point in the lattice and underspecification to mean a point higher in the lattice.[2] This is because overspecifying demand implies stating that more information is demanded than is really necessary when no precise representation for a demand exists. Similarly, underspecifying demand means stating that there is less "desire" for more information. Stated another way, underspecifying demand implies the information threshold enabling a greater amount of computation is higher.

Based on the definitions above, LD_{reuse} and ED_{reuse} are the lazy and eager variants, respectively, of an analysis overspecifying demand when a precise demand annotation is not available. These analyses are demarcated using the $reuse$ subscript because of their utility in making reuse decisions. Any time a function application is reached whose arguments are compliant with $reuse$ subscripted use annotations of a previously analyzed application, but are not compliant with the corresponding $reuse$ subscripted demand analysis, analysis of the new application is guaranteed to

[2]This is the opposite definition as for use.

produce the same results and annotations as those of the previously analyzed application. There is no point in performing symbolic evaluation of an application use compliant with a previously analyzed application unless it is also demand compliant with that application. The *reuse* subscripted demand analyses allow for optimization of the analysis phase through elimination of redundant symbolic execution of function applications for which an equivalent analysis has already been performed.

It is not obvious what value, if any, the ? subscripted demand analyses based underspecification serve. Clearly, two specializations of the same function for which the arguments to one application are compliant with the $ED_?$ of the other will produce non-identical code for any naive, straight forward code generator. However, the value of this observation is not immediately clear.

Applying Demand Analysis

If eager demand analysis for reuse is performed during symbolic execution, then analysis can be reused any time the information available about all arguments in a new application falls within the range delimited by eager use analysis for reuse and eager demand analysis for reuse as shown in Figure 5.8. None of the earlier descriptions of how the reuse optimization would be implemented or its general effects is changed by the expansion of the applicability of the optimization enabled by demand analysis. The only net change is that increased applicability implies greater gains from the optimization.

In closing, performing eager use and demand analyses for reuse does not necessarily obviate the need or desirability of performing the corresponding lazy versions of these analyses. Lazy reuse analysis is needed to generate the most efficient code during the code generation phase. Lazy analyses classify a broader set of conditions under which the same potential specialization can be utilized for multiple different call sites than do eager analyses. This greater reuse may yield tighter residual code than is possible with only eager versions of the analysis.

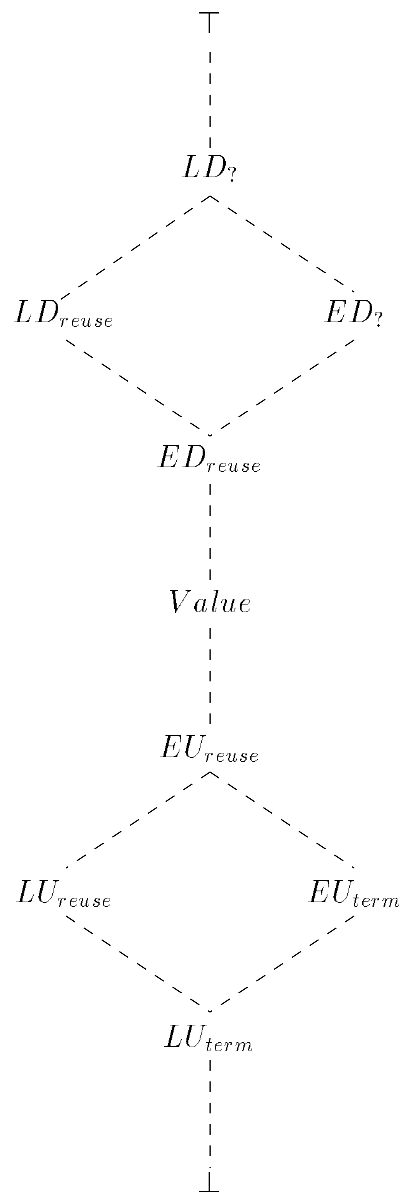

Figure 5.8: Relative amounts of demand and use information recorded by different analyses

5.2.2 Static Analysis

A possible means of reducing both the memory consumption and computational complexity of lazy use analysis is the utilization of some type of static analysis prior to lazy use analysis. Annotating single reference variables is one very simple example of this approach. A ternary BTA partitioning all values into static, dynamic, and unknown (e.g., [43]) might be used to reduce the number of variables needing to be considered during lazy use analysis, without modifying the semantics of lazy use analysis in any way. Dynamic variables never need to be considered in the analysis. The differentiation between purely static and unknown might also prove useful. Another approach is to perform a conservative, static version of lazy use analysis. Static approximations to use and demand might prove useful in reducing the complexity of performing dynamic analysis.

5.3 Conclusion

Large resource consumption, particularly memory, presents a serious obstacle to utilization of lazy use analysis as the basis of a production partial evaluator. Those approaches taken so far to reduce resource consumption have been insufficient to solve the problem. One, or both, of the untested ideas, or some new approach, is needed in order to make lazy use analysis a viable partial evaluation tool. In the absence of a true solution to the resource problem, it is possible some approximation to lazy use analysis might be identified that is more computationally feasible, yet produces nearly as good results.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Effective termination remains one of the most significant challenges for the partial evaluation community. Deciding when to perform additional analysis and when to terminate impacts many aspects of partial evaluation. Aggressively performing too much analysis can lead to divergence. Even when a partial evaluator terminates, analyzing too many potential specializations can yield overly verbose and therefore poor quality residual code. A conservative approach to termination, striving to minimize divergence, also often yields poor residual code due to the failure to create enough specializations. Creation of the correct, finite set of specializations is the key.

The motivation behind my work is the desire to develop a partial evaluator producing a better combination of reliable termination and quality residual code. I began my work by identifying a set of simple examples for which all existing partial evaluators produced suboptimal results in at least one case. As described in Chapter 4, the examples test a partial evaluator's ability to achieve all of the following: complete execution of expressions in order to produce a result value during partial evaluation whenever this is possible, termination of recursions exhibiting changing dynamic values under static control whenever the loops cannot be executed to completion during partial evaluation, and avoidance of premature termination of analysis leading to poor quality residual code of the form exhibited by many partial evaluators for programs like the regular expression matcher shown in chapter 4.

In order to achieve leverage on the dual goals of reliable termination and high quality residual code, I propose in Chapter 1 separating partial evaluation into two phases: analysis and code generation. This partitioning virtually eliminates the problem of poor quality residual code due to creation of too many specializations. It achieves this end by decoupling a decision to investigate a potential specialization during the analysis phase from the decision to generate code for a potential specialization during the code generation phase. The termination question is thereby limited to ensuring a sufficient number of potential specializations are analyzed while still terminating on a broad enough class of programs. Too much analysis is no longer a problem except from the standpoint of how long a partial evaluator takes to execute.

I frame the termination problem in terms of ensuring only a finite number of potential specializations are investigated by a partial evaluator in Chapter 2. In one form or another this is the underlying goal of every termination mechanism. I suggest using equivalence classes as a framework for categorizing potential specializations and deciding when to analyze different potential specializations. So long as the number of equivalence classes is finite and only a finite number of potential specializations are investigated for each equivalence class, the total number of potential specializations analyzed is also finite. All that remains is to define some finite set of equivalence classes and develop an algorithm for determining to which equivalence class each potential specializations belongs. Of course, while many different sets of equivalence classes may result in termination, not all yield high quality residual code.

The equivalence classes utilized in this work are based on use analysis. I claim in Chapter 2 that use analysis captures a fundamental characteristic of potential specializations, differentiating amongst them based on inherent semantic content. The equivalence classes are based on the information utilized in creating a potential specialization, not the information available to the specializer as is the case for many other termination mechanisms. As such, I argue the equivalence classes evolving from use analysis are not only effective for termination, but also yield potential specializations resulting in high quality residual code.

The initial concept of use of information by symbolic execution in creating potential specializations led to the broader concept of use of information by a program in performing computation, as represented by lazy use analysis. While eager use analysis is an approximation to the context free domain of specialization (CF-DOS), lazy use analysis is an approximation to the context sensitive domain of specialization (CS-DOS). Context sensitivity enables a finer distinction between different potential specializations based not only on the specializations created, but also on the context within which each is utilized. Different contexts use the results of applying a potential specialization in different ways. Differences in the uses of results further limit those aspects of a potential specialization of importance in a given context and therefore of fundamental interest in deciding equivalence with respect to that context.

Lazy use analysis initially suffered from premature termination. The equivalence classes produced were too large. Too many potential specializations were members of the same equivalence classes. As a result, an insufficient number of appropriate potential specializations were investigated.

The addition of base case analysis to lazy use analysis as presented in Chapter 2 appears to have solved the problem of premature termination for the small set of programs tested. Base case analysis is grounded in the observation that all terminating recursions must eventually execute a base case. Consequently, if a recursion in a residual program is to terminate at runtime, the partial evaluator must produce residual code for the base case. In order to produce residual code for a base case, a partial evaluator must analyze the base case. The contrapositive of this statement is, if a partial evaluator has not investigated a base case for a recursion, it cannot produce terminating residual code for the recursion. A direct consequence is that premature termination has taken place if a partial evaluator has not investigated at least one base case for each recursion.

Base case analysis detects those cases in which use analysis has terminated a recursion prior to investigation of at least one base. The results of base case analysis are utilized to cause additional symbolic execution to be performed of prematurely terminated recursions. Additional iterations of prematurely terminated recursions are investigated until at least one base case is analyzed. While base case analysis does not eliminate all cases of premature termination, it appears to go a long way towards mitigating this problem. A good compromise between effective termination and residual code quality appears to result.

The major shortcoming of lazy use analysis as described in Chapter 5 is the large amount of memory utilized in analyzing even very small programs. As currently implemented, lazy use analysis is not a practical termination mechanism on which to base a partial evaluator for optimizing real world programs. The reasons for the large resource consumption and some possible solutions to the problem appear in Chapter 5. Implementing and testing some of the untested approaches presented in that chapter remain a first critical step in the future development of use analysis. Unless the resources required for lazy use analysis can be drastically reduced, this analysis will remain a laboratory curiosity. Its contribution will largely be any influence it has on the future direction of research on termination.

At the moment, effective termination remains a core challenge that must be surmounted before partial evaluators can move from being research curiosities to widely utilized, productive tools. Neither the approach described herein nor any other work published has transitioned the state of the art to a point where partial evaluation can realistically become a part of standard, production compilers. Leaving aside the question of whether users are willing to accept compilers potentially failing to terminate, at the present time, partial evaluators do not produce a sufficiently good combination of high enough quality residual code and frequent termination in order to justify their widespread use.

6.2 Future Work

The first piece of future work is reimplementation of lazy use analysis with an eye towards reduction of resource consumption. The untested ideas presented in Chapter 5 ought to serve as a starting point for this process. Due to the global nature of the algorithms involved, unless an implementation can be found the resource consumption of which is closer to linear than to exponential, lazy use analysis will never be of any practical use.

Assuming a more efficient implementation technique is found, the next step is the implementation of base case analysis. Once base case analysis is operational, the complete analysis phase ought to be able to be tested on a broad class of programs to ensure it performs as expected and desired. Since it has never been proven on what classes of programs lazy use analysis converges, it will be at this point that more, albeit still anecdotal, information can be collected.

The final major step is the implementation of a code generation phase utilizing the use information supplied by the analysis phase. Until a full partial evaluator generating executable residual code is completed, the benefits of both lazy use analysis and two phase partial evaluation remain incompletely substantiated claims. Of particular interest will be an investigation of how effectively lazy use analysis can be utilized in order to allow specializations to be called from contexts most other analyses would not allow. Specializations not appearing to be correctness preserving based on the CF-DOS may none the less be correctness preserving and recognized as such based on the CS-DOS.

Even if better implementation technology is developed, it is unlikely the result will be a lazy use analysis executing in roughly linear resource consumption in the size of input programs. Consequently, another important avenue of future investigation is likely to be identifying more efficiently computed approximations to lazy use analysis yielding nearly as good termination properties and residual code.

Finally, use analysis was designed with an eye towards imperative languages. While the current formalization cannot handle imperative constructs, one of the motivations for use analysis was the fact it only characterizes potential specializations based on the subset of supplied information used during symbolic execution. For an imperative language, much of the heap is often a possible source of information during symbolic execution of a function, but only a very small portion of the information is actually used. Use analysis appears to offer leverage on the problem of efficiently deciding equivalence amongst applications in an imperative language. Of course, lazy use analysis of an imperative language would only exacerbate the resource problems from which the analysis already suffers. In addition, it would necessitate incorporation of other analyses, like alias analysis, many of which are themselves very computationally complex.

6.3 Off the Wall Thoughts by the Author

6.3.1 Partial Evaluation and Traditional Analyses

Lazy use analysis might be used as a means of characterizing different variants of functions for utilization by other analysis algorithms. The factor limiting precision of many analysis algorithms is the conflation of different variant uses of the same procedure. This happens in many of the standard compiler analyses presented in [2]. One possible use of lazy use analysis is as a means of distinguishing different variant uses of the same procedure so other analyses can separate out collection of information for the different variants and thereby produce more precise results.

6.3.2 Partial Evaluation and Optimization of Multilisp

Use analysis plus the partial evaluation framework is potentially applicable to some optimization problems that on the surface appear largely unrelated. At one point in the past I looked at the problem of automatically removing unnecessary *touches* of *futures* when compiling Multilisp [36], Halstead's multiprocessor version of Lisp. In Halstead's language, a future is a placeholder object into which a process stores its result once it is computed. Other processes can pass futures and store them in and retrieve them from data structures even before they are assigned values. However, a process attempting to utilize the value of a future before it is computed is suspended pending the calculation of the value.

Touch is the operation performed to check whether an argument is a future and if it is a future whether it has already been assigned a value. One of the major overheads of Multilisp is the need for every primitive function utilizing the value of one of its arguments to touch the argument prior to using it in a computation. To the extent a compiler is able to prove either that an argument cannot be a future or that the argument is a future with a known value, the overhead of futures can be greatly reduced.

I propose lazy use analysis and partial evaluation as a potentially effective approach to removing touches of futures. The domain of values of the partial evaluator would be expanded to include a non-future and a potentially future version for each type. Touch would become an operator coercing potentially future types to their respective non-future types. Partial evaluation would create appropriate versions of various primitives for future and non-future versions of their arguments. The partial evaluator would also create specializations of other procedures utilizing the appropriate versions of the primitives.

6.3.3 Deforestation and Driving

Wadler's work in deforestation [47] and Turchin's work in driving [46] are both approaches to fusing nested loops into single more efficient loops. With the exception of Turchin, no partial evaluator has been able to achieve this form of optimization. I believe the key to performing loop fusion in a standard partial evaluation framework is augmenting symbolic values so they include a richer representation of the computations needing to be performed at runtime to calculate the portions of values unknown during partial evaluation.

Weise's symbolic values are a combination of an abstract value and the residual code needed to compute the part of a value unknown until runtime. The residual code is the remainder of the source code not executable during partial evaluation. Alternatively, symbolic values could represent the part of the value known during partial evaluation and a more abstract representation of the computation needing to be performed at runtime in order to calculate the rest of the value at runtime. If the second part of a symbolic value were a more abstract description of a computation instead of code, I feel more powerful optimizations would be possible. During symbolic execution, representations of computations could be manipulated in ways not as easily performed with code. During the code generation phase, descriptions of computations would be converted into code, as opposed to the residual code for the return values simply being output.

While the details are obviously uncertain, I propose that something along the following lines might yield loop fusion. The domain of values of a partial evaluator would be extended to included recursive data types. When recursions were terminated during partial evaluation, the return values generated would be recursive data structures. In the case of nested loops, the inner loop would generate a recursive data structure as its result. The outer loop would consume this recursive data structure and produce another recursive data structure as its result. In many of those cases in which deforestation or driving produces a single loop, the recursive data structure produced for the outer recursion by the partial evaluator would be a simple recursive structure for which the code generation phase could create a single loop in the residual program.

Expanding the domain of values to include recursive data structures might have other advantages as well. Representing return values of recursions as recursive data structures, as opposed to flat data structures generated by generalization, might retain more information about the recursion for use in optimizing the continuation of the recursion. The additional information might enable better termination decisions to be made when analyzing the continuation, improving termination, residual code quality, or both.

Appendix A

Preprocessing of Factorial

This appendix shows how the three passes of the preprocessor transform the factorial program in Figure 2.17 on page 26 and repeated below.

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (-1+ n))))))
```

A.1 Pass 1: Result of the Front End

```
(define
 fact
 (lambda
  (named: (fact))
  (n)
  (if
   (application: (reference: not)
                 (application: (reference: zero?) (reference: n)))
   (application: (reference: *)
                 (reference: n)
                 (application: (reference: fact)
                               (application: (reference: -1+)
                                             (reference: n))))
   (constant: 1))))
```

A.2 Pass 2: Result of CPS conversion

```
(define
 fact
 (lambda
  (named: (fact))
  (n cont-328)
  (application:                                    ;(zero? n)
   (reference: zero?)
   (reference: n)
   (clambda
    (named: (cont-arg-864 fact))
    (cont-arg-864)
    (application:                                  ;(not (zero? n))
     (reference: not)
     (reference: cont-arg-864)
     (clambda
      (named: (cont-arg-856 fact))
      (cont-arg-856)
      (application:
       (lambda                                     ;Lambda introduced by
        (named: (cont-329 fact))                   ;CPS conversion
        (cont-329)
        (if                                        ;(if (not (zero? n)) ... )
         (reference: cont-arg-856)
         (application:                             ;(-1+ n)
          (reference: -1+)
          (reference: n)
          (clambda
           (named: (cont-arg-861 fact))
           (cont-arg-861)
           (application:                           ;(fact (-1+ n))
            (reference: fact)
            (reference: cont-arg-861)
            (clambda
             (named: (cont-arg-859 fact))
             (cont-arg-859)
             (application:                         ;(* (fact -1+ n) n)
       (reference: *)
       (reference: n)
       (reference: cont-arg-859)
       (clambda
        (named: (cont-331))
        (cont-arg-858)
        (exit-conditional                   ;Return (* (fact -1+ n) n)
         (clambda                           ;from conditional
          (named: (cont-330))
          (cont-arg-857)
          (throw (reference: cont-329)      ;Return (* (fact -1+ n) n)
                 (reference: cont-arg-857))) ;from introduced lambda
         (reference: cont-arg-858)))))))
  (application:
   (clambda
    (named: (cont-331))
    (cont-arg-858)
    (exit-conditional                       ;Return 1 from conditional
     (clambda
      (named: (cont-330))
      (cont-arg-857)
      (throw (reference: cont-329)          ;Return 1 from introduced
             (reference: cont-arg-857)))    ;lambda
     (reference: cont-arg-858)))
   (constant: 1))))
(clambda                                    ;Argument to lambda
 (named: (cont-328 fact))                   ;introduced by CPS
 (cont-arg-855)                             ;conversion
 (throw (reference: cont-328)               ;Return result of fact
        (reference: cont-arg-855)))))))))
```

A.3 Pass 3: Result of Alpha Conversion

```
(define
 (fact 86)
 (lambda
  (named: (fact))
  (formals: (n cont-332))
  (inheriteds:
   ((local-reference: * 29 ())
    (local-reference: fact 86 ())
    (local-reference: -1+ 41 ())
    (local-reference: not 42 ())
    (local-reference: zero? 21 ())))
  (application:                                               ;(zero? n)
   (inherited-reference: zero? 4 #t)
   (local-reference: n 0 ())
   (clambda
    (named: (cont-arg-876 fact))
    (formal: (cont-arg-876 0))
    (inheriteds:
     ((local-reference: cont-332 1 #t)
      (local-reference: n 0 ())
      (inherited-reference: * 0 #t)
      (inherited-reference: fact 1 #t)
      (inherited-reference: -1+ 2 #t)
      (inherited-reference: not 3 #t)))
    (application:                                             ;(not (zero? n))
     (inherited-reference: not 5 #t)
     (local-reference: cont-arg-876 0 #t)
     (clambda
      (named: (cont-arg-868 fact))
      (formal: (cont-arg-868 0))
      (inheriteds:
       ((inherited-reference: cont-332 0 #t)
        (inherited-reference: n 1 #t)
        (inherited-reference: * 2 #t)
        (inherited-reference: fact 3 #t)
        (inherited-reference: -1+ 4 #t)))
```

```
(application:
 (lambda                                                   ;Introduced lambda
  (named: (cont-333 fact))
  (formals: (cont-333))
  (inheriteds:
   ((local-reference: cont-arg-868 0 #t)
    (inherited-reference: n 1 #t)
    (inherited-reference: * 2 #t)
    (inherited-reference: fact 3 #t)
    (inherited-reference: -1+ 4 #t)))
  (if                                                      ;Conditional
   (inherited-reference: cont-arg-868 0 #t)
   (application:                                           ;(-1+ n)
    (inherited-reference: -1+ 4 #t)
    (inherited-reference: n 1 ())
    (clambda
     (named: (cont-arg-873 fact))
     (formal: (cont-arg-873 0))
     (inheriteds:
      ((local-reference: cont-333 0 ())
       (inherited-reference: n 1 ())
       (inherited-reference: * 2 #t)
       (inherited-reference: fact 3 #t)))
     (application:                                         ;(fact (-1+ n))
      (inherited-reference: fact 3 #t)
      (local-reference: cont-arg-873 0 #t)
      (clambda
       (named: (cont-arg-871 fact))
       (formal: (cont-arg-871 0))
       (inheriteds:
        ((inherited-reference: cont-333 0 #t)
         (inherited-reference: n 1 #t)
         (inherited-reference: * 2 #t)))
       (application:                                       ;(* (fact (-1+ n)))
        (inherited-reference: * 2 #t)
        (inherited-reference: n 1 #t)
        (local-reference: cont-arg-871 0 #t)
        (clambda
        (named: (cont-335))
        (formal: (cont-arg-870 0))
        (inheriteds:
         ((inherited-reference: cont-333 0 #t)))
        (exit-conditional                         ;Return (* ... )
         (clambda                                 ;from conditional
          (named: (cont-334))
          (formal: (cont-arg-869 0))
          (inheriteds:
           ((inherited-reference: cont-333 0 #t)))
          (throw                                  ;Return (* ... )
           (inherited-reference: cont-333 0 #t)   ;from introduced
           (local-reference: cont-arg-869 0 #t))) ;lambda
         (local-reference: cont-arg-870 0 #t))))))))
  (application:
   (clambda
    (named: (cont-335))
    (formal: (cont-arg-870 0))
    (inheriteds:
     ((local-reference: cont-333 0 ())))
    (exit-conditional                             ;Return 1 from
     (clambda                                     ;conditional
      (named: (cont-334))
      (formal: (cont-arg-869 0))
      (inheriteds:
       ((inherited-reference: cont-333 0 #t)))
      (throw                                      ;Return 1 from
       (inherited-reference: cont-333 0 #t)       ;introduced
       (local-reference: cont-arg-869 0 #t)))     ;lambda
     (local-reference: cont-arg-870 0 #t)))
   (constant: 1))))
(clambda                                          ;Argument to
 (named: (cont-332 fact))                         ;lambda introduced
 (formal: (cont-arg-867 0))                       ;by CPS conversion
 (inheriteds:
  ((inherited-reference: cont-332 0 #t)))
 (throw                                           ;Return result
  (inherited-reference: cont-332 0 #t)            ;of fact
```

```
        (local-reference: cont-arg-867 0 #t))))))))))
```

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techiniques, and Tools.* Addison-Wesley Publishing Company, Reading, Mass., 1986.

[3] Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher order functional language. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of the Third International Symposium on Static Anlysis*, pages 67–82. Springer-Verlag, 1996.

[4] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. In Neil D. Jones, editor, *Proceedings of the Third European Symposium on Programming*, pages 70–87, Copenhagen, Denmark, May 1990. Springer-Verlag Lecture Notes in Computer Science (432).

[5] Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[6] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991. Special issue on ESOP'90, the Third European Symposium on Programming, Copenhagen, Denmark, May 1990.

[7] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California. (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. ACM, 1992.

[8] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[9] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

[10] William Clinger and Jonathan Rees, *et. al.* Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4(3):1–55, 1991.

[11] Charles Consel. New insights into partial evaluation: the SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming*, pages 236–246. Springer-Verlag, LNCS 300, 1988.

[12] Charles Consel. *Analyse de programmes. Evaluation partielle et Génération de compilateurs.* PhD thesis, LITP, University of Paris 6, France, June 1989. In French.

[13] Charles Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[14] Charles Consel and Olivier Danvy. For a better support of static data flow. In J. Hughes, editor, *Conference on Functional Programming Languages and Computer Architecture*, pages 496–519, Cambridge, MA, 1991. Springer-Verlag (LNCS 523).

[15] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *Proceedings of the IEEE International Conference on Computer Languages*, Oakland, California, 1992.

[16] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

[17] Manuvir Das and Thomas Reps. Bta termination using cfl-reachability. Computer Science Department Technical Report 1329, University of Wisconsin-Madison, Madison, Wisconsin, 1996.

[18] Danny De Schreye and Stefaan Decorte. Terminiation of logic programs: The never-ending story. *Journal of Logic Programming*, 19(20):199–260, 1994.

[19] Michael J. Fischer. Lambda-caculus schemata. *LISP and Symbolic Computation (Special Issue on Continuations (Part 1))*, 6(3/4), December 1993. An earlier version appeared in an ACM conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No.1, January 1972.

[20] Richard P. Gabriel. The why of y. *Lisp Pointers*, 2(2):II–2.15–II–2.25, 1988.

[21] Torben Ægidius Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. Elsevier Science Publishers B. V., 1988.

[22] Arne J. Glenstrup and Neil D. Jones. Bta algorithms to ensure termination of off-line partial evaluation. In Andrei Ershov, editor, *Second International Conference 'Perspectives of System Informatics'*. Springer-Verlag, LNCS, 1996.

[23] R. Glück. Towards multiple self-application. In Jouannaud, editor, *Partial Evaluation and Semantics Based Program Manipulation*, New Haven, Connecticut, 1991.

[24] Chris Hanson, the MIT Scheme Team, and a cast of thousands. Mit scheme reference manual, edition 1.1 for scheme release 7.1.3. Technical report, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1991.

[25] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Conference on Functional Programming Languages and Computer Architecture*, pages 448–472, Cambridge, MA, 1991. Springer-Verlag (LNCS 523).

[26] Carsten Kehler Holst. Finiteness analysis. In J. Hughes, editor, *Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991. Springer-Verlag (LNCS 523).

[27] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jouannaud, editor, *Conference on Functional Programming and Computer Architecture*, Nancy, France, 1985. Springer-Verlag (LNCS 201).

[28] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[29] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. Elsevier Science Publishers B. V., 1988.

[30] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[31] Morry Katz and Daniel Weise. Towards a new perspective on partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 29–37, San Francisco, CA, 1992. Yale University Department of Computer Science. Published as YALEU/DCS/RR-909.

[32] Nathaniel David Osgood. Particle: An automatic program specialization system for imperative and low-level languages. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1993.

[33] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[34] J. C. Reynolds. Gedanken - a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13:308–319, 1970.

[35] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

[36] Jr. Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(5):501–538, 1985.

[37] Erik Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Stanford, CA, March 1993.

[38] Erik Ruf and Daniel Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, CN, June 1991.

[39] Erik Ruf and Daniel Weise. Preserving information during online partial evaluation. Stanford Computer Systems Laboratory CSL-TR-92-517, Stanford University, Stanford, California, April 1992.

[40] Peter Sestoft. Automatic call unfolding in a parial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–505. Elsevier Science Publishers B. V., 1988.

[41] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, October 1988. DIKU-Rapport 89/11.

[42] Morten Heine Sørensen. Turchin's supercompiler revisited. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1994. DIKU-rapport 94/17.

[43] Michael Sperber. Self-applicable online partial evaluation. In Olivier Danvy, Robert Gluck, and Peter Thiemann, editors, *Partial Evaluation International Seminar (1996: Dagstuhl Castle)*, pages 465–480. Springer-Verlag (LNCS 1110), 1996.

[44] Guy L. Steele. Rabbit: A compiler for scheme (a study in compiler optimization). Artificial Intelligence Laboratory AI-TR-474, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. An earlier version appeared as Steele's 1977 PhD dissertation from MIT.

[45] V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. Elsevier Science Publishers B. V., 1988.

[46] Valentin Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[47] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[48] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online program specialization. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 165–191. Springer-Verlag Lecture Notes in Computer Science, August 1991.

[49] Daniel Weise and Erik Ruf. Computing types during program specialization. Technical Report CSL-TR-90-441, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990. Updated version available as FUSE-MEMO-90-3-revised.