Topics in Online Partial Evaluation*

Erik Ruf

Technical Report: CSL-TR-93-563
(also FUSE Memo 93-14)

March, 1993

Computer Systems Laboratory
Departments of Electrical Engineering & Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Partial evaluation is a performance optimization technique for computer programs. When a program is run repeatedly with only small variations in its input, we can profit by taking the program and a description of the constant portion of the input, and producing a "specialized" program that computes the same input/output relation as the original program, but only for inputs satisfying the description. This program runs faster because computations depending only on constant inputs are performed only once, when the specialized program is constructed. This technique has not only proven useful for speeding up "interpretive" programs such as simulators, language interpreters, and pattern matchers, but also encompasses many "traditional" compiler optimizations.

Contemporary work in partial evaluation has concentrated on "offline" methods, where high-speed specialization is achieved at the cost of slower specialized programs by limiting the sorts of decision-making that can occur at specialization time. This dissertation investigates "online" methods, which impose no such restrictions, and demonstrates the benefits of specialization-time decision-making in FUSE, a partial evaluator for a functional subset of Scheme. We describe two new methods, return value analysis and parameter value analysis, for computing more accurate estimates of runtime values at specialization time, enabling more optimizations, and yielding better specialized programs with less "hand-tweaking" of the source. We develop a re-use analysis mechanism for avoiding redundancies in specialized code, improving the efficiency of both the specializer and the programs it produces. Finally, we show how to produce efficient, optimizing program generators by using our techniques to specialize an online program specializer.

Key Words and Phrases: Partial Evaluation, Program Specialization, Online Specialization, Abstract Interpretation, Control Flow Analysis, Polyvariant Specialization, Re-use Analysis, Program Generation

*This research was supported in part by NSF Contract No. MIP-8902764, by Advanced Research Projects Agency, Department of Defense, Contract No. N0039-91-K-0138, and by an AT&T Bell Laboratories Ph.D. Scholarship.

Preface

This report is a slightly reformatted version of the author's Ph.D. dissertation in Electrical Engineering at Stanford University. The work described here was performed during 1990-92 as part of the FUSE partial evaluation project under the direction of Professor Daniel Weise. Much of the work described in Chapters 2, 4, 5, 6, and 7 is reported in Stanford Computer Systems Laboratory technical reports [116, 96, 97, 93, 95], while work described in the ACM PEPM'91, FPCA'91, and PEPM'92 conference proceedings [92, 115, 94] appears in Chapters 3, 5, and 6. Portions of Chapter 6 are taken verbatim from the PEPM'91 article [92]; grateful acknowledgment is made to the ACM for permission to republish them here.

Acknowledgments

Looking back over my graduate career, I have many people to thank.

Daniel Weise's compiler course kindled my interest in program transformation using Scheme, and he gave me the freedom to choose my own research topics when everyone else wanted me to conform to their goals (usually involving FORTRAN or C++). Daniel's enthusiasm and support helped keep me interested in research when things weren't going very well. His departure is truly Stanford's loss and Microsoft's gain.

Carolyn Talcott acted as a surrogate advisor to our entire research group after Daniel left Stanford. I thank her for her very careful reading of my dissertation, and for help in formalizing my algorithms. Teresa Meng also read the dissertation and provided good advice about my future plans.

AT&T Bell Laboratories made this work possible by supporting me for the last four years. Not having to search for new funding each year contributed greatly to my peace of mind, and, I hope, to the quality of this research.

The Embedded Computation Area at Xerox PARC provided not only a quiet place for me to work, but also a stimulating intellectual environment. My fellow "reading group" members Michael Ashley, Jim des Rivières, Mike Dixon, Gregor Kiczales, John Lamping, and Luis Rodriguez gave me plenty of constructive criticism. John Lamping deserves special recognition for wading through all four of my major technical reports.

My fellow students Morry Katz and Scott Seligman were my co-explorers not only in partial evaluation, but also in the Stanford bureaucracy, HP-UX system administration, and MIT Scheme. As fellow "Daniel survivors," their conversation and camaraderie was most helpful.

My friends David Eliezer, Mark Goldner, Steve Keifling, Bob Lodenkamper, Kirk Miller, Mike Yang, and especially Mitchell Trott and Barbara Sinkule, helped preserve my sanity during this project. I thank them for providing much-needed distraction from work and for listening to all my gripes.

Finally, my parents, Ernst and Rosmarie Ruf, helped start all this back in 1987 by encouraging me to move to Palo Alto instead of Schenectady. Seems they were right. I wish them the best of luck in their new life in Zurich.

Contents

List of Figures

Chapter 1

Introduction

Program specialization, also referred to as *partial evaluation*, is a program transformation technique used for improving the performance of computer programs. The work described in this dissertation makes several contributions to the technology of program specialization, and also hopes to contribute to a better understanding of the benefits and limitations of existing program specialization methods.

This chapter introduces program specialization and gives a brief history of work in the field. It also describes the contributions made by this work, and concludes with an outline of the dissertation.

1.1 Introducing Program Specialization

A program specializer transforms a program and a partial description of its inputs into a new, *specialized* program which, when applied to any input satisfying the description, computes the same result as the original program. The specializer uses the information in the specification to perform some of the program's computations at specialization time, resulting in a specialized program that runs faster than the original program did.

Program specialization is most useful when applied to general-purpose programs which are run repeatedly with part of their input held constant. If we specialize the program on the constant portion of the input, we can perform the portion of the computation depending on that input once, at specialization time, rather than over and over again on each invocation of the program. The tradeoff, of course, is that the specialization process takes time, so we must execute the specialized program enough times so that the cumulative time savings

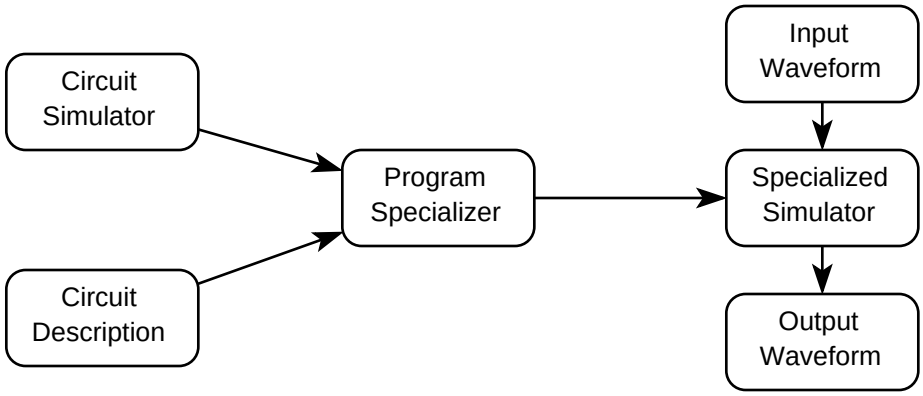


Figure 1.1: Specializing a circuit simulator on a particular circuit

is greater than the cost of constructing the specialized program. This research focuses more on the production of efficient specialized programs than on the efficient production of specialized programs, though both issues will be addressed to some degree.

As a real-world example, consider a program for simulating analog electrical circuits. It takes two inputs: a circuit description and a set of initial conditions, and produces one output: a description of the circuit's state over time. A circuit designer typically simulates a particular circuit on many different sets of initial conditions. Some of the simulator's computations can be performed given only the circuit description; *e.g.,* there is no need for the simulator to "rediscover" the topology of the circuit each time the initial conditions are changed. Other computations, such as the computation of an individual circuit element's output from its input, cannot be performed until the actual input values are available.

This usage pattern suggests that we could benefit by using a program specializer to specialize the circuit simulator on its first input, the circuit description, then run the specialized program on many different sets of initial conditions (*c.f.* Figure 1.1). This experiment was performed by Berlin and Weise [10], who found that specializing the simulator on a circuit produced specialized programs that ran up to 91 times faster than the original circuit simulator.[1] We can make two important observations here. First, program specialization is

[1] Not all of this speedup is directly due to the incorporation of the circuit description into the specialized program. Some of the overhead in the simulator is due to the use of abstractions which can be removed even without knowledge of the circuit description.

more appropriate for some program/input combinations than for others. If we were to specialize the circuit simulator on its second input, the initial conditions, we wouldn't expect much improvement at all, because without the circuit description, the simulator doesn't know what to *do* with the initial condition values. Typically, program specialization is applied to *interpretive* programs where one of the inputs *controls* the computation to be performed, while the other inputs are merely the *subjects* of the computation [63]. Other examples include interpreters for programming languages (specialized w.r.t. the program to be interpreted), pattern matchers (specialized w.r.t. the pattern to be located), and object-oriented dispatchers (specialized w.r.t. the class hierarchy, message, and receiver type, but not w.r.t. the argument values).

Second, program specialization can provide significant performance improvements even when no input description is provided. This is because, even though the program itself cannot be specialized, general-purpose subprograms within the program often can be specialized, removing layers of both procedural and data abstraction. Consider the C function `sprintf`[2] (or the LISP function `format`), which prints a series of data to a string based on formatting directives contained in a string argument. Such a function is essentially a small interpreter that uses one argument (the format control string) to determine how the other arguments are processed. Furthermore, a typical program contains many invocations of `sprintf`, often with constant format control strings. A program specializer can create a specialized version of `sprintf` for each of its invocation sites, eliminating the need to parse format control strings at runtime. Similarly, higher-order functions such as `map` and `compose` can be specialized on arguments which are literal `lambda` expressions, eliminating the need for closure creation at runtime. Loops with known bounds can be unfolded, type checks performed, and intermediate data structures eliminated.

Indeed, program specialization can be viewed as a number of "traditional" compiler optimizations such as constant folding, constant and copy propagation, dead code and intermediate data structure elimination, coupled with a flow-sensitive interprocedural analysis and liberal amounts of inlining and procedure cloning [31] to take advantage of that sensitivity. In theory, the only difference is that a program specializer is willing to operate on information (concerning the constant inputs) which is not present in the program, but

[2] We use `sprintf` rather than its more common cousin `printf` because the notion of "printing to a string" is easier to express as a functional input/output relationship. The argument for `printf` is similar.

rather is specified by the user. Practically speaking, the advantage of present-day program specialization over most traditional optimization techniques is one of aggressiveness: most specializers are willing to build multiple specialized versions (up to one per call site) of a single procedure to achieve a high degree of optimization, rather than building only one version which must be sufficiently general to be used at all call sites, as in traditional interprocedural approaches.

1.2 Brief History

Program specialization has a long history; we cannot cover all historical contributions in this dissertation. Contributions relating to particular portions of this dissertation will be addressed in the "Related Work" sections of individual chapters. We will attempt to give a brief survey of program specialization work for functional languages here.[3] For alternate treatments of this material, we suggest [63] or [28].

The roots of program specialization can be traced back to the S-m-n theorem [73], whose proof gives a construction for specialization of Turing Machines, though not a very efficient one. The earliest program specializer we know of, for LISP programs, was presented in 1964 [78]. Other early work in the 1960's and 1970's on program specializers for LISP [7, 49, 67] stressed the optimizing properties of program specialization, and resulted in the development of symbolic type systems, conditional contexts, and conditional handling of side effects. Another powerful specialization technique, *supercompilation*, also developed at this time, was not widely known until the 1980's [111].

A common application of this early work was the "compilation"[4] of programs by specializing interpreters. In the 1970's, Futamura [40] and Ershov [39] independently discovered the principle of compiler generation via *self-application*, in which the specializer is specialized on an interpreter, yielding a program with the functionality of a compiler, but which is more efficient than direct specialization of the interpreter. Later, it was discovered that even

[3] Program specialization has also been pursued widely in the logic programming community; [38, 100, 74, 107] are only a few examples. Some progress has also been made for imperative languages; see [4, 80] for references.

[4] In our view, specializing an interpreter performs only a portion of the function of a compiler. The process does convert a source program into a program which, when given the program's input, produces its output, and which thus has the same type signature as an object program. However, much of the complexity of "real" compilers lies in dealing with resource allocation issues, such as register allocation, memory management, and the like, which have not been addressed by interpreter-based program generation techniques because it's difficult to expose such issues in an interpreter.

this process could be optimized by specializing the specializer on itself, yielding a "compiler compiler" [37]. These uses of the specializer for compilation, compiler generation, and compiler-compiler generation have come to be known as the "Futamura Projections."

Subsequent research stressed the efficiency of specialization via this self-application property. The first self-applicable system, Mix [64, 65], was soon followed by a host of other, more powerful self-applicable program specializers [12, 83, 22, 23, 76, 45]. All of these systems achieved self-application through the use of a particular implementation technique called *offline specialization*, in which some of the specializer's decisions[5] are made by a preprocessing phase called Binding Time Analysis (BTA). This allowed the specializer to be kept quite small and efficient while preserving the information necessary for successful self-application [15, 12]. Even without self-application, BTA has been shown useful for gaining efficiency [25, 29] and for adding various other functionality to the program specializer [14]. One cost of this method was a loss of optimization, as the BTA pass introduces approximations that need not be made by non-offline systems.

In the late 1980's, a resurgence of interest in "optimization over efficiency" motivated new research into *online* specialization techniques, which in many ways bear more similarity to the techniques of the 1960's and 1970's than to the post-Mix generation of program specializers. Work on such methods includes the MIT scientific computing project [10, 8, 9], the Stanford FUSE project [115], and others [46, 101, 43, 118]. At least one online specializer [43] has achieved self-application, though with heavy use of offline techniques (see Chapter 7 for details).

At present, work on program specialization continues on several fronts. Both the offline and online communities have made recent gains with respect to the quality of the specialized programs they produce [26, 29, 55, 98, 92, 94], the language constructs which they can specialize [45, 4, 117], and the termination [54, 70] and efficiency [52, 77, 95] properties of the specialization process. The usefulness of program specialization has been demonstrated in increasingly wide domains of examples, including inheritance [71], simulation [6, 117], incremental computation [110], and operating systems [30]. Theoretical work on basic properties of specialization [5, 79, 48] and Binding Time Analysis [62, 76, 59] continues as well. Increasingly, one also finds program-specialization-like techniques being used in more

[5] Those relating to deciding which computations to perform at specialization time and which to defer until runtime via the generation of specialized code. We call these decisions *reduce/residualize* decisions, and will treat them in greater detail in later chapters.

traditional compilation disciplines, such as customized compilation [19, 18] and procedure cloning [31].

1.3 This Work

The main premise behind this dissertation, as with other work on online specialization, is that the primary goal of program specialization is to construct efficient specialized programs from realistic, minimally "hand-tweaked" source programs. That is, efficiency of specialized programs is our primary concern, while efficiency of the specializer is of only secondary importance.

To this end, we studied the output of existing specializers, especially early versions of FUSE [114, 115], and noticed several deficiencies:

- **Sub-optimal Specializations, Idiom Sensitivity**: The specialized programs produced by the specializer failed to contain optimizations that we expected would be performed given the input specifications provided by the user. Furthermore, the degree of optimization varied greatly when the source programs were changed only slightly.

- **Redundancy**: The specialized programs sometimes contained multiple specialized procedures that were identical, modulo renaming. This not only led to larger residual programs which took longer to compile, etc., but also slowed the specialization process because of the time needed to construct the extra specializations.

- **Inability to construct Program Generators**: Under offline methods, a common tactic for increasing the efficiency of specialization is to construct a "program generator" by specializing a specializer.[6] The construction of efficient program generators based on online specializers was not believed to be possible.

We will address these problems by investigating both online and offline methods, and will show that some optimizations are expressed more easily in an online framework. From there, we proceed to extend the "state of the art" of online techniques, both in the direction of producing better specialized programs, and in the direction of specialization-time

[6] We distinguish "program generator generation," where a specializer is specialized, from "self-application," where a specializer is specialized *using the same specializer*. We will demonstrate the former, but not the latter.

efficiency. We will show that online techniques are practical, powerful, and efficient, as well as implementable, by describing their use in our program specialization system, FUSE. Our contributions, as expressed in this dissertation, include:

- A detailed analysis of the limitations of offline approaches, motivating the use of online techniques,
- Two new algorithms, *return value analysis* and *parameter value analysis*, for reducing information loss during the specialization of first-order and higher-order programs, resulting in better specializations of many programs written in "natural" styles,
- A description of the problem of redundant specializations, along with a *re-use analysis* algorithm for detecting potential redundancies before they occur, improving the efficiency of both the specializer and the programs it produces, and
- The first example of the generation of efficient program generators from a truly *online* program specializer without any use of binding time approximation techniques.

1.4 Outline

In Chapter 2, we define program specialization in greater detail. We introduce the distinction between *online* and *offline* specialization, and explain the limitations of the offline approach which motivated us to research online methods. Chapter 3 describes the implementation of our online program specializer.

The work in Chapters 4, 5, and 6 forms the core of the dissertation, giving improved specialization algorithms which improve the quality (speed) of specialized programs. In Chapter 4, we show how existing program specializers for first-order languages lose information unnecessarily by failing to compute return value estimates for specialized procedures, and describe the fixpoint iteration-based method used to solve this problem in FUSE. Chapter 5 treats a similar information loss problem for higher-order languages, namely the need to compute good approximations to the values passed as parameters to specializations of first-class functions, and describes FUSE's solution, which is based on control flow analysis. Chapter 6 introduces a different problem, that of redundant specializations, which affects the efficiency of both the specializer and the programs it produces; we describe a solution based on computing *use information* during the specialization process.

We then turn to an application: the generation of program generators by specializing specializers. It has long been believed that only specializers using explicit binding time approximations (*i.e.,* offline specializers, plus Glück's "online BTA" system [43]) could be specialized into efficient program generators. Chapter 7 shows otherwise, describing how we used FUSE to produce efficient program generators from a small online specializer (a subset of FUSE). Chapter 8 concludes the dissertation, describing our contributions and some open problems for future work. The Appendix gives more formal descriptions of the algorithms in Chapters 3, 4, 5, and 6, and the properties on which they rely.

A note to the reader: Chapters 4, 5, 6, and 7 are fairly independent, though they often refer to concepts introduced in Chapter 2 and implementation details introduced in Chapter 3. Those who are familiar with the tradeoffs between online and offline methods may wish to skip Chapter 2, while those who are familiar with or uninterested in online implementation techniques may wish to skip Chapter 3. The Appendix makes heavy use of terminology from Chapter 3. We assume familiarity with the Scheme [88] programming language and with basic concepts of programming language semantics.

Chapter 2

Online and Offline Program Specialization

Program specializers operate by symbolically executing a program on a specification of its inputs. During this process, the specializer decides to reduce (*i.e.,* execute at specialization time) some expression, and to residualize (*i.e.,* put in the specialized program, for execution at runtime) others. These decisions affect the quality (both speed and size) of the specialized program, as well as the efficiency and termination properties of the specializer itself.

In this chapter, we examine two widely-used methods for making these decisions: the *offline* framework, in which all *reduce/residualize* decisions are made statically before the specializer runs, and the *online* framework, in which at least some of these decisions are made dynamically during specialization. We motivate our investigations into online techniques by showing that, in certain cases, the offline method delays computations which could be performed at specialization time until runtime, thus producing slower specialized programs than the online method, which can perform these reductions at specialization time.

We begin, in Section 2.1, by defining program specialization in a general manner. However, this definition has two shortcomings: it does not indicate how to implement a specializer, nor does it say anything about the performance of the specialized programs that are produced. We therefore proceed to a more operational treatment: Section 2.2 describes the framework of *polyvariant specialization*, which is used by almost all specializers, while Section 2.3 describes the online and offline variants of polyvariant specialization. In Section 2.4, which forms the bulk of the chapter, we describe two primary limitations of offline approaches, how they affect the quality of specialized programs, and why online approaches

do not encounter these problems. We conclude with a survey of related work on improving the quality of specialization under both offline and online techniques.

In this dissertation, we will limit ourselves to describing the specialization of programs written in functional languages; our examples will be written in a functional subset of Scheme [88], occasionally extended with pattern-matching for brevity. Given this restriction, we will treat the terms "program," "function definition," and "procedure definition" as synonyms. To denote the amount of information the specializer is given about a value, we use the terms "known" and "unknown," as well as "partially known."[1]

2.1 Defining Program Specialization

A *specializer* takes a function definition and a specification of the arguments to that function, and produces a residual function definition, or *specialization*. The argument specification restricts the possible values of the actual parameters that will be passed to the function at runtime. Although different specializers use different specification techniques, thus allowing different classes of values to be described, they all share this same general input/output behavior.

We can define a specializer as follows:[2]

Definition 1 *(Specializer) Let $\mathcal{L}$ be a language with value domain V and evaluation function $E : \mathcal{L} \times V \to V$. Let S be a set of possible specifications of values in V, and let $C : S \to (\mathcal{PS}\ V)$ be a "concretization" function mapping a specification into the set of values it denotes. A* specializer *is a function $SP : \mathcal{L} \times S \to \mathcal{L}$ mapping a function definition $f \in \mathcal{L}$ and a specification $s \in S$ into a residual function definition $(SP\ f\ s) \in \mathcal{L}$ such that*

$$\forall a \in (C\ s)\,[(E\ f\ a) \neq \bot_V \Rightarrow\ (E\ f\ a) = (E\ (SP\ f\ s)\ a)]\,.$$

This definition has several important properties. First, it allows the residual function definition to return any value when the original function definition fails to terminate (indicated by the evaluator returning $\bot_V$); a stricter definition would preserve the termination

[1] As we will describe in Section 2.3.1, the more commonly used terms, "static" and "dynamic," are not properties of values, but are instead properties of expressions, meaning "will denote only known values at specialization time," and "may denote unknown values at specialization time," respectively.

[2] We use LISP-like prefix notation for all function applications in this dissertation; *i.e.*, $(f\ x\ y)$ denotes the application of f to parameters x and y.

properties of the original function definition. Second, the residual function definition takes the same formal parameters as the original function; we consider reparameterization [100] behavior such as the removal of completely known parameters or arity raising [90] to be a code generation issue, and not part of the definition of specialization. Finally, this definition gives a correctness criterion for specializers, but no information about how they operate, giving us no information about the performance of the residual function definition program relative to that of the original function definition. To describe and compare various strategies for performing program specialization, we must take a more operational view of specialization.

2.2 Polyvariant Specialization

Most program specializers operate by symbolically executing the program; for each redex, the specializer either performs a one-step reduction on the redex, or builds a residual code expression which will perform that reduction at runtime. The specializer repetitively makes this *reduce/residualize* decision for each redex (or *program point*) encountered during the symbolic evaluation.

This choice is what distinguishes a program specializer from an interpreter; ordinary evaluators and interpreters always reduce the current redex, while the specializer has the option of delaying reduction until runtime. Clearly, it is desirable for the specializer to perform as many reductions as possible to avoid the need to perform them when the residual program is evaluated. At first, this might seem simple: have the specializer perform all reductions except those prohibited by a lack of information. That is, build residual code only for `if` expressions with unknown tests, function applications (combinations) with unknown heads, and primitive expressions with unknown parameters in strict positions.

Unfortunately, such a naive strategy often fails to terminate due to infinite unfolding of function calls in loops. Overly-eager reduction of procedure calls may miss opportunities for sharing in residual programs, and risks duplicating computations via beta-substitution. Finally, representational issues may forbid certain reductions, such as those which would place specialization-time data structures in residual contexts (see [12] for a discussion of some of these issues).[3]

[3] Another approach to dealing with representational issues such as code duplication and specialization-time structures in residual code is to delay some reduce/residualize decisions until after specialization is

Thus, all specializers limit specialization-time reductions by performing *folding* [17] operations. Folding is done by recursively specializing certain distinguished parts[4] of the program with respect to their arguments, and replacing their invocations with residual invocations of the corresponding specialization; this is often called *program point specialization* [62]. These specializations are memoized (we often say "cached") in the specializer, so that they can be invoked from multiple call sites in the residual program; this allows for code sharing and also acts as a form of loop detection.

Strategies for building and caching specializations vary. Most specializers use a strategy called *polyvariant specialization* [16], in which multiple specialized versions, or *variants*, of a single program point may be constructed: program points are specialized with respect to the known values in their argument specifications, and cached specializations are re-used when all of the known values in their specifications match exactly. Some implementations of this approach, such as [49, 101, 116, 29] increase the number and specificity of specializations by building them based on known types of otherwise unknown values. Others, such as the system described in Chapter 6 and in [92, 93], decrease the number of specializations without loss of runtime performance via a more sophisticated caching mechanism.

For the specializer to terminate, it must ensure that all loops which cannot be completely executed by finite unfolding are broken by a residual call to a specialization, and must also ensure that only a finite number of such specializations are built. Choosing to residualize a function call provides the former, but to provide the latter, the specializer must often prohibit other reductions as well. For instance, consider specializing the function

```
(define (length l ans)
  (if (null? l)
      ans
      (length (cdr l) (+ 1 ans))))
```

on unknown `l` and `ans=0`. Under a naive strategy, choosing to fold the recursive call yields an infinite set of specializations, one for each non-negative integer value which `ans` can assume. In order to terminate, the specializer must choose to residualize the expressions `ans` and `(+ 1 ans)` even though they are reducible.

complete; see Chapter 3 and [114, 115] for details.

[4] In most specializers, these points are user function definitions, although some specializers, like Similix [14], have a prepass that adds additional function definitions to the program to enable specialization of program points that were not originally user functions.

It is vital that the specializer build specializations which are sufficiently general to be applicable in more than one context. Loops, as shown above, are one example: the same specialization must be applicable both at the initial and recursive entry points to the loop. Another example of the need for such general behavior occurs in higher-order programs, in which a closure created by a residual `lambda` expression must be applicable at multiple call sites. Consider specializing

```
(define (length2 l k)
  (if (null? l)
      (k 0)
      (length2 (cdr l)
               (lambda (ans)
                 (k (+ 1 ans))))))
```

on unknown `l` and `k`. Not only must the specializer fold the recursive call to `length2` and residualize both invocations of `k`, but it must also build a residual version of the continuation `(lambda (ans) ...)` which must be applicable at both invocations of `k`. The reason for building only a single residual version, as opposed to one per call site, is that both call sites of `(lambda (ans) ...)` are also reached by the initial continuation passed in to `length`, thus forbidding rewriting the call sites to invoke different specializations of `(lambda (ans) ...)`.[5] In building the specialization, the specializer may only use information which is common to to arguments at both call sites, in this case, the knowledge that the argument is an integer. The expression `(+ 1 ans)` must be left residual even though `ans` is known to be `0` in one of its invocation contexts.

Thus, making the reduce/residualize choice is not as simple as it might, at first, have seemed. Not only must a specializer only perform reductions when it has sufficient information to do so, but it must sometimes disallow reductions in order to build sufficiently general specializations, and to handle various code generation and representation issues. The goal is to build a sufficient number of specializations of sufficient generality without

[5] If we were to rewrite the program into a first-order form that passed environments explicitly and performed an explicit `case` dispatch among the various function bodies reaching each call site, we could indeed build specializations on a per-call-site basis. We view this transformation as overly low-level because it forces a user-level representation of a virtual machine object, the environment. Such transformations may be counterproductive because they limit the Scheme compiler's ability to choose efficient machine-level representations. Once first-class environments become part of the Scheme language, program specializers will have more options when specializing programs such as the one above. For a comparison of several higher-order specialization techniques, see Section 5.1.

foregoing reductions needlessly, which would increase the number of reductions performed at runtime.

2.3 Methods for Polyvariant Specialization

This section describes two different implementations of polyvariant specialization, the offline and online methods, which differ in how they make reduce/residualize decisions.

2.3.1 Offline Polyvariant Specialization

Description

Offline specializers are those which do not make reduce/residualize choices at specialization time. Instead, they make these choices prior to specialization, usually without full knowledge of the argument specification on which the specializer will be invoked. Along with the program and the argument specification, the specializer is given a set of directions (usually in the form of annotations on the program) which completely determine all the reduce/residualize choices it will make. Thus, unlike the naive strategy (which performs all possible reductions), an offline specializer will not examine the values of the subforms of an expression when deciding whether to reduce it. When the pre-ordained choice is "reduce," then it will, of course, use the values of subforms in performing the reduction, but it will never use such specialization-time information to dynamically choose whether to perform the reduction at all.

One way of describing this approach is to consider a reformulation of the source program in which each expression is annotated with either a "reduce" mark or a "residualize" mark. Marks on formal parameters indicate whether the corresponding actuals should be included in the key used to search for existing specializations in the cache. For instance, the `length` example above (annotated for unknown `l` and known `ans`) might be annotated as follows:

```
(define (length l ans)
  (if (null?  l)
      ans
      (length (cdr l)
              (+ 1 ans))))
```

where underlining denotes the "residualize" mark (residual applications have both parentheses underlined), while nonunderlined expressions are considered to be marked "reduce." Specialization proceeds via syntactic dispatch as in the naive strategy, but the reduce/residualize choice at each reduction step is made via the annotation on the corresponding expression. In our example, the instances of `null?`, `cdr`, and `+` will be reduced to primitive procedures, `length` will be reduced to a compound procedure, and everything else, including all procedure applications, will be residualized. Note that the residualization of the above code doesn't depend on the values of either `l` or `ans`; even if these values are known, they will be ignored. Annotating the formals `l` and `ans` as "residualize" conveys this information to the caching mechanism, ensuring that duplicate versions of the specialization won't be built for different values of the parameters.

Under such an annotation scheme, not all annotations of a program are considered well-formed. The specializer must be able to perform the reductions specified by the annotations. It should never be the case that an expression that might receive an unknown argument in a strict position is marked "reduce," nor the case that a formal parameter that could be bound to an unknown value is marked "reduce." For instance, if the expression `(null? l)` in the example were marked "reduce," the specializer would have to decide whether the unknown value bound to `l` is the empty list, which isn't possible. Similarly, if the formal `ans` were marked "reduce," the specializer would needlessly build separate specializations for the calls `(length foo 0)` and `(length foo 100)` with `foo` unknown. Well-formedness criteria for annotations have been developed as part of the work on "congruence" described in [62, 76].

Usually, these annotations are placed semi-automatically.[6] A prepass called Binding Time Analysis [64] (BTA) computes, for every expression, a conservative approximation to its specialization-time value, determining whether the expression is "static" (its value is guaranteed to be known at specialization time) or "dynamic" (its value might be unknown at specialization time).[7] These approximations are used to compute the "reduce" and "residualize" annotations: dynamic identifiers must be residualized, as must all special forms

[6] Binding Time Analysis and the placement of the "reduce" and "residualize" annotations have been a matter of much research, but will not be an issue in the remainder of this chapter; most of our conclusions are solely a function of the use of annotations, and are independent of the means used for placing them. We include this description solely for completeness.

[7] More sophisticated binding time analyses compute more detailed approximations, such as "the value will either be completely known (static) or will be a list of pairs, each of which has known (static) `car` and possibly unknown (dynamic) `cdr`." For details of such analyses, see [83, 22, 23, 76].

and primitive procedure calls with dynamic arguments in strict positions. Everything else may be marked "reduce." As we saw before, however, to provide folding and termination, certain calls (and, possibly, certain arguments to those calls) must be marked "residualize." Such additional annotation is usually performed via a variety of means, including manually (as in the original Mix [64] and the "generalization" operator of Similix [14]), automatically (the "dynamic conditionals" of Similix and the "finiteness analysis" of [54]), or via a meta-language (the "filters" of Schism [23]). The results of binding time analysis are also used by the memoization code in the specializer, which uses the binding time descriptions of the actual parameters to decide what to use as the search key (the usual practice is to use the "static" parts of the parameters, instead of annotating the formals as in our example above).

Motivation

The primary motivations behind the offline method are simplicity and efficiency of the specializer. Offline specializers can be made almost as small as interpreters, since, with the exception of memoization, all of the decisions made by specializers but not by interpreters have been performed at annotation time. Thus, an offline specializer typically requires no state beyond that an interpreter would maintain, with the exception of the cache of specializations.

Offline specializers can also economize with respect to the representation of values; the specializer only needs to represent those data values which will actually be used in the reductions it will make; in particular, there is no need to represent the "unknown" value, meaning that it may be possible to inherit most of the representation of values from the underlying system without any additional encoding.[8]

Finally, since an offline specializer's reduce/residualize behavior is completely determined by its program argument, but is valid for different specification arguments, it is possible to build a version of the specializer with the reduce/residualization behavior "built in," eliminating the need for interpreting the annotations. This has been accomplished to varying degrees by (1) moving more functionality into the annotations [25], (2) handwriting a compiler generator [56], and (3) self-applying the partial evaluator [40, 64]. Offline specializers are particularly amenable to self-application due to their small size, lack of an

[8] Some offline systems, particularly those handling higher-order functions or strongly typed languages, still require their own representations; see [77] and the description of *pe-closures* in [12].

encoding problem, and statically determined reduce/residualize behavior. Forcing the reduce/residualize behavior to depend only on the the program argument, which is known at self-application time, avoids any danger of losing that information at specialization time and turning a static choice into a dynamic one [15].

However, this specialization-time efficiency does have a cost in runtime performance (*c.f.* Section 2.4).

2.3.2 Online Polyvariant Specialization

Description

Online specializers are those which make reduce/residualize choices at specialization time. They do this much like one might imagine a "naive" program specializer operates: the specializer represents unknown values explicitly, so that it can determine whether `if`, application, and strict primitive expressions are reducible merely by examining their argument values. In addition, such a specializer needs some mechanism to force folding and the creation of sufficiently general specialization bodies.

The explicit representation of unknown values requires, at minimum, adding a distinguished "unknown" value to the type lattice of the language being partially evaluated. Usually, online specializers go further, extending the type system to allow unknown values to be placed in data structures and closures, and to provide unknown values with known attributes such as types or arithmetic signs [115, 29].

Online specializers use a variety of mechanisms to cause the building of specializations: some are based on explicit reduce/residualize annotations [43], others on a meta-language [21], and still others on various forms of recursion detection [115, 112]. The meta-language approach allows the user to define a "fold here?" predicate for each function, which is passed the function's arguments and decides whether to unfold or specialize. The recursion detection mechanisms compute call histories (essentially a representation of the pending procedure returns in the interpreter) and fold whenever a recursive call can't be proven to be well-founded. Systems with a fixed "fold here" annotation have folding behavior identical to that of offline specializers, which must mark certain calls as "residualize"; systems with more dynamic mechanisms will fold less often, leaving fewer residual procedure calls. Premature folding results in extra procedure calls at runtime, but isn't a major problem in and of itself. What *is* important is how, after making the decision to fold,

the specializer builds the specialization.

What distinguishes the specialization behavior of online specializers is how, after having decided to fold, they build suitably general instances of functions. Unlike offline specializers, which pre-compute which reductions to perform while building the general function instance, online specializers achieve generality by modifying the argument values on which the specialization will be built, thus indirectly affecting which reductions will be performed during the construction of the general instance. This process is usually called *generalization* [112, 115], but has also been referred to as *generalized restart* [100]. After deciding to build a specialization, the specializer finds the set of call sites for which the specialization must be applicable, then computes the least general argument specification which includes the argument specifications of all relevant call sites, and builds the specification using the new, more general, specification. Since argument specifications are just vectors of values, this generalization can be accomplished by computing least upper bounds in the domain of values. Generalization usually occurs in a *pairwise* manner; as new call sites are added to a specialization by the folding mechanism, the specification is recomputed, and the specialization is rebuilt (an implementation of this mechanism is described in Section 3.3.4). The important feature of online generalization mechanisms is that they only lose information (by computing more general specifications) when this is necessary; any information which is common across the call sites is retained. This allows, for instance, the maintenance of type information in loops with polymorphic parameters, and the building of minimally general specialized versions of first-class procedures with multiple call sites.

Aside: online specializers that really aren't

Note that one can build an "online" specializer that makes the same reduce/residualize choices as an offline specializer would by simply running a binding time analysis in parallel with the specializer. This approach would construct the "directions" for reduce/residualize decisions at specialization time, then obey them. One such a system, described in [43], performs BTA (called "configuration analysis" in [43]) on-the-fly before specializing each function. This analysis could have been performed statically by a polyvariant BTA; performing it dynamically achieves similar polyvariance with a simpler analysis.

We would expect that such systems would be efficiently self-applicable, since, just as in the offline case, all reduce/residualize decisions are made by a process that refers only to the program text and statically available information (binding times), so there is no danger

of losing this information at self-application time. Unfortunately, such methods have the same conservative behavior as purely offline methods, because they only use the "static" and "dynamic" approximations to known and unknown values when making reduce-residualize choices. Our further discussions of the online method will explicitly assume that such a "vacuously online" strategy is not in use. We will return to such techniques when we address self-application in Chapter 7.

Motivation

The primary motivation behind the online specialization method is that delaying the reduce/residualize choice until specialization time means that more information will be available when the choice is made, and thus the choice can be made more accurately. Efficiency considerations are considered secondary to the goal of performing as many reductions as possible at specialization time. Unlike the offline strategy, which performs generalization ahead of time, and can always build a specialization in a single pass, online specializers are willing to recompute specializations, possibly multiple times (as in Chapters 4 and 5, and [97, 94]), to build better residual programs.

The specialization-time efficiency cost is real. Providing an enhanced value domain, checking "known-ness" on each primitive reduction, keeping a context to decide when to fold, and computing general argument specifications (possibly multiple times) means that online specializers generally (1) are larger, (2) use more memory, and (3) take longer to run than their offline counterparts.

The next section describes areas where online specialization either constructs faster residual programs than offline specialization, or constructs similar programs with less need for "pre-transformation" of the source program.

2.4 Problems with Offline Methods

Program specializers operate by performing some of a program's computations at specialization time, producing a residual program that executes fewer operations (and thus runs faster) at runtime. Ideally, then, a program specializer should perform as many reductions as possible[9] (while still terminating) at specialization time, so that these reductions wouldn't

[9]This does not, unfortunately, give us an absolute metric for residual programs. Consider the case of a loop that cannot be fully unfolded at specialization time. A specializer can always perform one more

have to be performed at runtime. We use the term "accuracy of specialization" to connote a measure of how many reductions a specializer is able to perform at specialization time. More accurate specializers perform more reductions, and produce faster residual programs, while less accurate specializers miss opportunities for reductions, producing slower residual programs.

Offline specializers trade some amount of accuracy for (specialization-time) efficiency, while online specializers do the converse. In this section, we analyze several situations in which accuracy sacrificed by an offline specializer could be recovered by an online one.

The inaccuracies of offline specialization stem from two sources: the need to represent multiple specialization-time contexts with a single annotation on a single source expression, and the inability to take advantage of commonality in application contexts when building specializations. For the most part, these concerns are based solely on the property that offline specializers are directed by annotations, and are independent of the process for producing those annotations (manually or via binding time analysis). When a particular BTA strategy affects our argument, we will make this clear.

2.4.1 Representing Multiple Contexts with one Annotation

At runtime, a particular expression may be evaluated many times. Each time it is evaluated, its free variables may be bound to different values, causing it to compute a different value. Not only will the values obtained by performing reductions in the expression change, but, due to the use of conditionals, the set of reductions performed may vary as well.

This behavior also occurs at specialization time, but with another twist: unlike evaluation, where there is always enough information to reduce any redex, the set of reductions performed may also vary based on whether sufficient information is available to reduce them. Unfortunately, in the offline paradigm, a particular source expression is annotated as either "always reduce" or "never reduce" (residualize). The inaccuracy here is obvious, since all specialization-time instances of an expression must be treated in the same manner regardless of the circumstances, any expression which might be non-reducible in *some* context must be left residual in *all* contexts. At runtime, this causes a penalty in the form of

reduction at specialization time by unfolding the loop one more time (thus eliminating a procedure call at runtime, at the cost of increasing the code size). This is always the case; no matter how many times the source loop is unfolded before the residual loop is constructed, unfolding it one more time will always produce a "better" residual program under such a metric.

extra reductions that could have been performed at specialization time. Worse yet, failing to reduce an expression doesn't affect that expression alone: its return value will be lost, and any other expression that requires that value to be reduced will become irreducible as well.

The problem is that even though a particular expression can be reduced to many different values by a polyvariant offline specializer, it (and its subforms) will always be reduced in the same *manner*. One approach to alleviating this problem, called polyvariant binding time analysis, creates separate contexts by duplicating code in the source program; this works in many situations, but is limited because the amount of duplication must be statically determined without knowledge of the size or values of data that will be known at specialization time. Context can also be duplicated via continuation-passing-style (CPS) conversion of the source program; this works well in some cases, but can still fall prey to the problems faced by polyvariant BTA.

Another approach to the problem doesn't attempt to perform the reductions at specialization time, but instead makes local reductions (reducing `if`s with known tests and beta-reducing applications with known heads which aren't part of loops, etc.) in a post-processing phase. This recovers some of the lost efficiency by performing reductions before runtime, but doesn't regain all of it. In most cases, the postpass is not as powerful as a specializer; in particular, it lacks the ability to build new specializations based on information made available by the reductions it performs. To perform all possible reductions either requires a postpass with the full power of a specializer[10], or requires iterating specialization. Such iteration has been suggested, but is not commonly performed because it would require running both the specializer and the annotation phase multiple times. Not only would this negate much of the efficiency advantage of offline specialization, but it would make systems with manual or human-assisted annotation schemes difficult to use, due to the need to annotate the intermediate machine-generated programs.

In this section, we describe three situations in which the necessity to represent multiple contexts with a single annotation leads to less efficient residual programs, and describe how the online approach avoids these inaccuracies. The first such situation involves computing the reduce/residualize annotation for expressions that use the return value of a conditional

[10]Note that such a postpass would be an *online* specializer, and, as such, would work just as well without having its input pre-specialized.

expression which is reducible at specialization time. The second situation occurs in interpreters, which often contain expressions whose reducibility depends on specialization-time data, while the third situation involves computing annotations for expressions which access data structures whose size is unknown until specialization time. All three of these situations pose no problem for online specializers because they will make the reduce/residualize decision at specialization time, when sufficient information is available for the choice to be obvious.

Return Values from Specialization-time Choices

One case where contexts are known at specialization time, but not before, occurs when an expression returns one of two values, one known, one unknown, based on a choice which will be known at specialization time. An example of this behavior is an `if` expression with a known test and one known arm, such as

```
(if (= a 0) b c)
```

annotated for `a` and `b` known at specialization time, and `c` unknown. This forces `c` to be annotated as residual, but allows everything else to be reduced. So far, so good. However, consider a case in which this `if` returns its value to some other expression, such as:

```
(+ (if (= a 0) b c) 5)
```

If the `if` returns `b`, the specializer can reduce the application of `+`; if it returns `c`, it can't. Since the annotation must be performed *without knowledge of* `a`*'s value*, we can't predict which arm the `if` will return. Thus, the application of `+` must be annotated as residual even though it might be reducible:

```
(+ (if (= a 0) b c) 5)
```

If `a=0` is the usual case, the residual program will perform many `+` reductions which could have been performed at specialization time. This can get worse: any expression surrounding the `+` would also have to be left residual; this sort of "chaining" can lead to large numbers of unnecessary residualizations.

An offline specializer's performance can be improved by transforming the input program; such transformations are often called "binding time improvements." One method, due to

Mogensen [83], distributes outer computations across inner ones. Such duplication of source code allows each context to be considered independently. In this example, the program would become

```
(if (= a 0) (+ b 5) (+ c 5))
```

which would be annotated as

```
(if (= a 0) (+ b 5) (+ c 5))
```

The two copies of the application of `+` can each be annotated separately, leading to a more accurate annotation, and thus a more accurate specialization.

Another method, due to Consel and Danvy [26], performs CPS conversion on the program, yielding

```
(let ((k (lambda (temp) (+ temp 5))))
  (if (= a 0)
      (k b)
      (k c)))
```

which duplicates context by creating two entry points to the continuation instead of just one. The specializer can treat these entry points separately either by unfolding both invocations of `k`, or by building a different specialization at each call site. Unfortunately, CPS conversion introduces higher-order constructs, which can be difficult to specialize. Many systems use monovariant higher-order BTA, under which the expression `(+ a temp)` would not be duplicated, and would be marked residual, giving no improvement.[11] Consel and Danvy address this problem by duplicating the text of continuations to `if` expressions (rather than just `let`-binding the continuation) at CPS conversion time, allowing the copies to be annotated separately.

Both of these "context duplication" strategies fail when such an `if` is embedded in a loop. Consider

```
(iterate loop ((i i) (a a) (acc 0))
  (if (= i 0)
      (1+ acc)
      (loop (- i 1) (- a i) (if (= a 0) b c))))
```

[11] This is merely a problem with common BTA strategies, not a problem with static annotation or the CPS strategy in general.

with `i`, `a`, `b` known and `c` unknown. Even though the loop can be completely unfolded at specialization time, no amount of code duplication prior to specialization (and annotation) will be able to build a context in which `acc` (and thus the return value of the loop) is guaranteed to be known; thus, the call to `1+`, and any consumer of the loop's return value, must be left residual, even though `acc` may always be known.

Similar problems arise if various algebraic optimizations are performed: with `a` known and `b` unknown, the expression

```
(* a b)
```

may return a known value if `a=0`, so it's incorrect to assume that any expression using the return value must be residualized. Admittedly, this sort of optimization is rare in arithmetic, but if one considers error values, then operations on partially known inputs may often return known values.

The online strategy handles all examples of this variety trivially, since it waits until specialization time, at which point the inner expression's return value is computed, to make the reduce/residualize decision for the outer expression.

Data-Dependent Contexts in Interpreters

Another example of a single expression representing several specialization-time contexts arises in the context of interpreters, simulators, or other programs in which the program repetitively traverses one input while performing computations on the other. The static nature of annotations requires that all traversals have identical reduce/residualize behavior, even though the specialization-time values may be different on each iteration.

This effectively prohibits the specializer from taking advantage of structure in the program being interpreted, yielding a "non-optimizing" compilation process. Consider a fragment from an interpreter for a trivial higher-order language with unary functions and binary primitive operators, as shown in Figure 2.1.

The single expression `(lookup e1 env)` implements all variable references in the program; similarly, the single `if` expression in the interpreter implements all of the `if` expressions in the program, and the expression `(eval (fcn-body fcn) ...)` in `apply` implements all of the applications in the program.

Consider annotating this program (actually, its entry point, `eval-pgm`) for `lambda-exp` known and `actual` unknown. Because `actual` is unknown, the call to `lookup` may return an

```
(define (eval-pgm lambda-exp actual)
  (eval (lambda-exp-body lambda-exp)
        (make-env formal actual)))

(define (eval exp env)
  (case exp
    ([const e1]           e1)
    ([var e1]             (lookup e1 env))
    ([if e1 e2 e3]        (if (eval e1 env)
                              (eval e2 env)
                              (eval e3 env)))
    ([lambda formal body] (make-fcn formal body env))
    ([apply e1 e2]        (let ((fcn (eval e1 env))
                                (arg (eval e2 env)))
                            (apply fcn arg)))
    ([primop p arg1 arg2] (p (eval arg1 env) (eval arg2 env)))))

(define (apply fcn arg)
  (eval (fcn-body fcn)
        (extend-env (fcn-env fcn)
                    (fcn-formal fcn)
                    arg)))
```

Figure 2.1: A sample interpreter for higher-order programs. If `env` does not bind all program identifiers to known values, then `fcn`, `arg`, and `exp` could become bound to unknown values. Under a static annotation scheme, all operations referring to these variables must be marked as residual, meaning that even syntactic dispatch cannot be resolved at specialization time. The problem is less severe in interpreters for first-order languages because `exp` will always be known.

unknown value at specialization time. Thus, any call to `eval` may return an unknown value at specialization time. This means that the expression `(if (eval e1 env) ...)` must be left residual; we cannot reduce any of the program's `if` expressions at specialization time, even if they are known (as is often the case when interpreting programs with initialization code). Worse yet, `fcn` and `arg` may become bound to unknown values (because the recursive calls to `eval` may return unknown values), so the `exp` argument to `eval` may become bound to an unknown value, so the `case` statement must be left residual. This means we can't even reduce the syntactic dispatch of the interpreter at specialization time!

The binding time improvement techniques used above don't work here, as they rely on duplicating expressions to duplicate context. In this case, we need to duplicate the appropriate arm of the `case` for every expression in the program being interpreted; this isn't possible at annotation time because the program isn't available yet. Bondorf [11] solves the problem by cleverly rewriting the interpreter to represent interpreter closures as Scheme closures (*i.e.,* replace `(make-closure formal body env)` with `(lambda (arg) (eval body (extend-env env formal arg)))` and rewrite `apply` appropriately), effectively moving the recursive call to `eval` from `apply` into `make-closure`. This trick keeps the syntactic parameter `exp` from becoming dynamic, but doesn't allow the reduction of `if`s (or, for that matter, function applications) at specialization time. Accomplishing such reductions requires iterating specialization, as described above.

Once again, an online specializer (in principle[12]) has no problems with this example because it makes its reduce/residualize decisions at specialization time, when the necessary information is available.

Aggregates

Static annotations complicate reasoning about aggregates, such as lists and sets, whose sizes or element values are not known until specialization time. At specialization time, the elements of a list may be completely independent of one another; some may be known and others unknown. Given a loop (say, the function `map`) traversing a list and performing some operations on the elements, we would like to perform the operations on the known elements

[12] Existing online specializers have no problem reducing `if`s and known function applications in the program being interpreted, but will generate sub-optimal code when unknown interpreter closures are applied unless their representation of values includes disjoint union types. Thus, Bondorf's method is still of use in the online case.

and leave them residual on the unknown elements; furthermore, in the case of a list with an unknown tail, we would like to unfold the loop until that tail is reached, and then build a residual loop.

These behaviors cannot always be achieved under a static annotation strategy. The decision of whether to reduce or residualize the elementwise operations must be encoded on the source program, which is only possible if we know the length of the list and the known/unknown status of each element of the list at annotation time; otherwise, we must annotate all of the elementwise operations as "residualize," losing all knowledge of the known elements. This is similar to the interpreter case described in the previous section (page 24), in which we were unable to distinguish the environment binding of `actual`, which was unknown, from other bindings which might be known.

The desired result of unfolding a loop until the unknown tail is reached is also difficult to achieve under a strategy where the "unfold vs. specialize" decision is made per static call site rather than per dynamic call instance. If one iteration of the loop is unfolded, all iterations will be; the loop would have to be pre-unfolded based on *annotation-time* knowledge of its length. Deciding on specialization instead of unfolding is less detrimental in systems which can compute return values of residual function calls, because such systems could still return a representation of the list, with the known (processed) values. Unfortunately, returning such a representation once again requires annotation-time knowledge of the list's length to pre-duplicate the code which traverses the representation of the returned list.

This situation is common in systems using worklists, such as circuit simulators and abstract interpreters; it also occurs in the context of interpreters with nonempty initial environments (there's no reason to declare the bindings for primitives and library functions as unknown just because later frames may bind variables to unknown values).

In some cases, this behavior might be adequate if the annotation strategy could handle lists whose construction is completely determined by the source program, even if it couldn't handle lists built under static control at specialization time. An example of such a structure might be one which doesn't depend on any data supplied at specialization time, such as an initial environment frame or other initial state built by an interpreter or simulator. Current methods for computing annotations don't take advantage of concrete information (constants, etc) in the program being annotated; these get abstracted just like the inputs do. This is one reason why specializing a program on completely unknown inputs (thus performing reductions based on concrete values in the program), then recomputing the

annotations, can be helpful.[13]

2.4.2 Inability to take advantage of Commonality

We have seen examples of where the offline strategy of static annotation forces overly conservative behavior by requiring residualization of an expression in all contexts if any of its contexts require residualization, thus (1) failing to make reductions at specialization time, and (2) failing to compute information which would enable reductions in other expressions.

We now describe a different form of conservative behavior, which is tied to the residualization behavior of specializers. Any residual expression built by the specializer must be sufficiently general to compute the correct result for any runtime context under which it is invoked. Specializers accomplish this by maintaining conservative approximations of the values which might be returned by the expressions in the program at runtime, and only perform a particular reduction at specialization time when the approximations indicate that it is the only runtime possibility.

Often, the specializer is forced to build a single approximation which represents several possible runtime executions. Doing this may require merging existing approximations (*i.e.,* computing a single approximation which denotes at least the union of the denotations of the approximations being merged; typically, the approximations are taken from a type lattice, and merging two approximations is equivalent to computing their least upper bound.). For example, the value returned from a residual `if` expression could be the value of either arm; similarly, the value of a formal parameter of a specialization could be the corresponding actual parameter from any of the specialization's call sites.

Because generality forces more reductions to occur at runtime, we would like to build the least general residual expression which is still sufficiently general. One obvious method is to discard only that information which differs between contexts, and to use the (remaining) common information in performing specialization. This is the motivation behind the generalization strategy often used in online specializers. Unfortunately, finding this sort of commonality requires testing whether specialization-time values (and their subparts) are equal. Such testing isn't explicitly outlawed by the offline method—even offline polyvariant

[13] This limitation is due to present annotation methods, and is not an indictment of annotation general. It is worth noting, however, that any annotation method capable of using such concrete information would be very similar to an online specializer, in that it would have to make duplication, residualization, and termination decisions dynamically as it runs.

specializers compare static argument values in the cache to achieve re-use of specializations and thus folding. However, using the results of such an equality test for the purpose of making a reduce/residualize decision *is* prohibited. That is, an offline specializer may not make use of any value obtained by merging others unless it can prove, at annotation time, that all of the values being merged will be equal at specialization time. If this were not the case, the specializer would have to examine the merged value (to determine if it was known or unknown) to decide whether to perform reductions depending on it. In many cases, it is not possible to perform such a proof at annotation time; even in cases in which it might be possible to perform such a proof, all current annotation methods fail to do so, and thus all values returned by residual calls to specialized procedures, or passed as parameters to residual first-class procedures are always declared to be "dynamic," forcing residualization of all references to them.

The offline approach to solving this problem relies on program transformations that duplicate code to reduce the number of contexts in which a particular residual expression must be applicable; if the number of contexts can be reduced to one, then no commonality testing is necessary. Another strategy attempts to convert "upward" commonality involving return values, which can't be compared, into "downward" commonality involving parameter values, which can be compared.

In this section, we describe three cases in which the inability to compute a sufficiently specific generalization leads to a loss of information at specialization time, and describe how the online approach avoids this problem. The first case involves deciding whether to reduce or to residualize expressions that use the return value of a residual conditional expression. The second case treats the use of return values of specializations, while the third case involves computing the parameter approximations used to build specializations.

Return values of `if` expressions

Perhaps the simplest case where generalization occurs is when an `if` is not decidable at specialization time. In such cases, the specializer's approximation of the value returned by the `if` must approximate the runtime return values of both arms. Consider the program

```
(... (if (= a 0) b c) ...)
```

where `a` is unknown, and both `b` and `c` are known. If `b` and `c` have the same value, then we can reduce the `if` expression to that value, and use it in reducing the enclosing expression;

otherwise, we must leave the `if` residual, and can only use information common to the values of both `b` and `c` in reducing the enclosing expression. Of course, it's unlikely that the variables `b` and `c` will have the same value, but it is often the case that their values will share some common properties. For instance, programmers often construct "ad hoc" types by constructing tagged pairs; it is worthwhile to take advantage of the equality of tags in these situations. For example, if `b=(foo-type . <unknown>)` and `c=(foo-type . <unknown>)`, returning `(foo-type . <unknown>)` instead of `<unknown>` will allow the specializer to reduce type tests of the form `(eq? (car (if (= a 0) b c)) 'foo-type)`.[14] Similarly, in specializers which compute types or other static properties of unknown values [116, 29], one would like to retain type information common to both branches of a residual `if` expression. For instance, it might be useful to know that both arms of an `if` are integers when reducing an enclosing application of `+`.

Unfortunately, it is often difficult, or impossible, to determine at annotation time whether two values, their subparts, or their types, will be equal at specialization time. In our example above, it would be necessary to prove, not only that `b` and `c` will have known types at specialization time, but that these types will be the same. In some cases, this might be provable; in others, in which the equality of the types depends on data not available until specialization time, such a proof would be impossible. Thus, an equality test is often necessary at specialization time; if the types (or values) are equal, then they are returned; otherwise, the result is unknown. This sort of conditional "known-ness" based on equality tests is effectively forbidden under the offline paradigm because the reduce/residualize decision for an expression consuming such a "conditionally known" value would have to be made by examining the value at specialization time. If such a value could be unknown, it must be annotated as dynamic, preventing its use in any reductions.

The context duplication methods (code copying and CPS conversion) described in Section 2.4.1 work in this case, because they distribute consumers of conditionally known values across the conditional. For example,

```
(foo (if a b c))
```

[14] At first, it might appear that such reductions are useful only in programs exhibiting "Lisp hacker" programming style, and that static type inference would be sufficient to perform such reductions in languages with better data abstraction facilities, such as statically typed languages. This is not the case: even in a typed language, an interpreter (or, in the case of self-application, a program specializer) for a runtime-polymorphic language must resort to ad hoc typing, due to the need to represent values in the interpreted program using a single universal type.

is transformed to either

```
(if a (foo b) (foo c))
```

or

```
(let ((k (lambda (x) (foo x))))
  (if a (k b) (k c)))
```

However, unlike the previous case in which the `if` was reduced to one of its arms at specialization time, in this example, both arms will appear in the residual program, so context duplication will result in duplication of code. In cases where the known portions of the two arms differ, this is indeed desirable, as it will allow all of the information in both arms (rather than just the information common to both) to be used in reducing the application of `foo`, but in cases where the known portions are identical, this duplication will serve no useful purpose. This naturally suggests duplicating the context only when the generalization loses information; such an optimization is possible under an online strategy, but not under an offline one due to the necessity of making reduce/residualize decisions based on a specialization-time comparison.[15] As was the case before (*c.f.* Section 2.4.1), loops can make it impossible to perform such context duplication statically; we will see an example of this in the next section.

Online methods handle both the normal and CPS-converted versions of this example well, due to their willingness to base reduce/residualize decisions on comparisons between specialization-time values. Information common to both arms of the conditional is preserved, and used in performing reductions in the surrounding context, while information which differs is discarded, causing residual accessor code to be generated.

Return values of specializations

Another example of context sharing occurs at residual call sites: the expression(s) surrounding the call must be able to handle any values returned by the call. For each residual call

[15] To date, no online specializer implements such an "on-the-fly" duplication strategy; the point is that such an optimization is possible only in an online framework. Offline specializers can achieve this behavior via postprocessing: Similix-2 [11] handles this case by forcing all `lambda`-bodies to be specialization points (in this case, building either one or two specializations of `k`, depending on whether `c=d`), then unfolding some of those specializations in a postpass (if `c=d`, the specialization will not be unfolded, keeping sharing; otherwise, both specializations will be unfolded).

to a specialization, the specializer must decide which of the surrounding expressions are reducible, and which must be residualized. One simple tactic assumes that all references to the return value of a residual call must themselves be left residual; this is obviously safe, but misses reductions not only at calls to the specialization, but, in the case of specializations of recursive functions, inside the specialization itself. Better approximations can be returned if generalization is performed, but this requires performing comparisons which are forbidden under the offline scheme.

Our first example involves the preservation of type information during generalization. Consider the "iterative" factorial function

```
(define (fact n ans)
  (if (= n 0)
      ans
      (fact (- n 1) (* n ans))))
```

specialized on `n` and `ans` specified as unknown integers.[16] We would like to retain the information that `n` and `ans` remain integers on the recursive call, and that the return value is an integer. Retaining the type of `n` and `ans` is simple: an annotation mechanism can prove that their types are static; thus, the type information will be retained when the specialization is built, potentially allowing the use of `=`, `-`, and `*` operators with fewer tag checks. No annotation-time knowledge of type equality is necessary, because the specializer will simply build a new variant if the types of `n` or `ans` change on the recursive call.

In an offline specializer, retaining the type information of the return value is slightly more difficult. To prove that the return value has a "static" type, the annotation phase must prove that the types of `ans` and `(* n ans)` are the same. For integers, this is easy to prove, but the annotation phase must be able to prove this knowing only that the types of `n` and `ans` are "static," not that they are integers. Unfortunately, that proof is not possible (the specification admits `n` a float and `ans` an integer; in that case, the type of `ans` is "dynamic").

In this case, the type information for the return value can be preserved by duplicating context, building a new version of `fact` per call site. We could transform the program

[16] For brevity, we assume a specializer which reasons about typed unknown values. This is not required; the same commonality problems arise in the presence of "ad hoc" user types built out of untyped data structures, but their use would significantly enlarge this example.

```
(letrec ((fact (lambda (n ans)
                 (if (= n 0)
                     ans
                     (fact (- n 1) (* n ans))))))
  (+ (fact a b) 99))
```

into either

```
(letrec ((fact (lambda (n ans)
                 (if (= n 0)
                     (+ ans 99)
                     (fact (- n 1) (* n ans))))))
  (fact a b))
```

or

```
(letrec ((fact (lambda (n ans k)
                 (if (= n 0)
                     (k ans)
                     (fact (- n 1) (* n ans) k)))))
  (fact a b (lambda (temp) (+ temp 99))))
```

In either case, the specializer would now be able to deduce that `99` is added to an integer, and simplify the `+` operator accordingly. No annotation-time proofs are necessary, because now there is no need to compute a generalization at each `if`, since any computations depending on the `if`'s return value have been moved into its arms. Both of the above solutions duplicate code, the first because it has incorporated the continuation at each call site (in this case, the addition of 99) into the definition of `fact`; the second, because each call site will pass a different continuation, leading to the creation of a different specialization per call site. This is unfortunate, since every one of `fact`'s non-recursive call sites will get its own specialization, even if all of them call `fact` on the same arguments!

Both the "context distribution" and CPS conversion solutions require that `fact` be tail-recursive. The copying solution requires that the expression returning the loop's final value be inside the loop, which is not the case in a truly recursive program. Similarly, the CPS solution unfolds the continuation argument to duplicate context; in a recursive version of factorial, in which the continuation is extended on each iteration, termination would require that the continuation parameter be made dynamic, at which point the type of its argument would be lost.

Another way of viewing this is to note that both of these techniques work by converting an upward-passed value into a downward-passed value to avoid losing information

at a point where approximations must be merged (the `if` statement). Information about downward-passed values is never lost, because specializations are shared only when their static arguments match exactly; different values result in a new specialization instead of a more general one. Unfortunately, for truly recursive programs expressed in CPS style, termination often requires that multiple call sites with different static arguments share the same specialization, necessitating online comparisons and reduce/residualize decision making to retain information. We will revisit this problem on page 38.

An online specializer can handle these return values quite naturally; instead of transforming the program to avoid the problem (and thus introducing other problems), if it is willing to build an initial approximation to the return value, then weaken it later as necessary. For details of how this is accomplished, see Chapter 4 or [116, 97].

To show that this behavior occurs in contexts other than scalar type inference, our second example comes from the domain of interpreters. The code fragments in Figure 2.2 implement part of an interpreter for a "toy" imperative language. This interpreter represents the store as a list of pairs, each of whose `car` is an identifier and whose `cdr` is its value.

We can "compile" a statement by specializing the function `eval-stmt` on a known statement and an initial store containing known identifiers and unknown values. We would like the specializer to determine that the "shape" of the store parameter never changed; that is, it will always contain the name known identifiers, in the same order, as the store on which `eval-stmt` was initially specialized. Retaining this information allows variable lookups to be reduced to simple list accesses[17] without any need for comparing identifiers. There should be no residual loops which search for bindings in the store.

Consider specializing `eval-while` on a particular `while` loop. The calls to `eval-expr` and `eval-stmt` can be unfolded, but the `if` expression and the recursive call to `eval-while` must be left residual. To obtain the desired result, the specializer must be able to deduce that the store returned from the `if` (and thus from the residual call to `eval-while`) has the same shape as the store on which `eval-while` was specialized. If this correspondence is lost, and the return value is approximated by "unknown," then any variable accesses or updates in statements following the `while` statement will be implemented by residual loops.

Just as in the case of the factorial example, an online specializer gives the desired result because it can compare stores, discover that their shapes are the same, and return

[17] Later arity raising can provide further simplification by converting the list structure representation of the store into separate variables.

```
;;; eval-stmt: stmt x store -> store
(define (eval-stmt stmt store)
  (case stmt
    ([if e1 s2 s3]  ...)
    ([:= e1 e2]    (update store e1 (eval-expr e2 store)))
    ([begin . s]   (eval-begin s))
    ([while e1 s2] (eval-while stmt store))
    ([call e1]     (eval-call e1 store))
     ...))

;;; eval-expr: exp x store -> value
(define (eval-expr exp store) ...)

;;; eval-begin: stmt* x store -> store
(define (eval-begin stmts store)
  (if (null? stmts)
      store
      (eval-begin (cdr stmts) (eval-stmt (car stmts) store))))

;;; eval-while: stmt x store -> store
(define (eval-while stmt store)
  (if (eval-expr (while-test exp) store)
      (eval-while stmt (eval-stmt (while-body stmt) store))
      store))

;;; find-proc-body: procname x pgm -> stmt
(define (find-proc-body proc-name pgm) ...)

;;; eval-call: procname x store -> store
(define (eval-call proc-name store)
  (let ((proc-body (find-proc-body proc-name pgm)))
    (eval-stmt proc-body store)))
```

Figure 2.2: Fragments from a direct-style MP interpreter

a generalized store upward. Offline specializers, which cannot perform such comparisons, cannot handle the interpreter directly; this is a recognized open problem in offline partial evaluation [76]. There are several transformational approaches to solving this problem. One approach separates the store into two lists: one which holds the names, and need only be passed downward, and one which holds the values, and is passed both upward and downward. Now even if upward-passed values are approximated by "unknown," the list of names will not be lost. Of course, any information about the values in the store (such as their types) will be lost. Such rewriting is often performed by hand, but automated versions have been constructed [83, 34]; these have the disadvantage of being able to preserve only values which can be proven, at annotation time, to be known at specialization time; any data-dependent information will be lost.

Another approach rewrites the program so that it only passes the store downwards, never upwards. This approach is taken by Mogensen [83] and by Launchbury [76], whose interpreters pass an extra argument containing the "rest" of the MP+ statements to be executed. Instead of returning a store when the loop is complete, `eval-while` exits by calling `eval-stmt` on the "next statement" and the store. This approach involves considerable hand-rewriting, and may not generalize well to other programs.

The third common work-around, which works well for interpreters for languages with iteration but without recursion, is to perform the CPS transformation on the program, as is done by Consel and Danvy [26]. After this transformation is applied, the store is passed as a parameter to continuations instead of being returned upward, eliminating the need for accurate approximations to returned values.

Unfortunately, the CPS transformation fails to preserve the shape of the store when we specialize the interpreter of Figure 2.2 on a program containing recursive procedure calls. In this case, the residual interpreter will contain continuations (residual `lambda` expressions) which must be applicable from multiple call sites with different known argument values; retaining the information common to their call sites cannot be accomplished by binding time improvement techniques, because the equality of the known argument values, which is needed at annotation time, cannot be tested until specialization time. We will discuss this problem further in the next section. Online methods handle programs with procedure calls without difficulty, because they can preserve the shape of the store directly by computing generalized return values as necessary.

Parameters of specializations

The examples thus far in this section have dealt with merging approximations of values which are returned upward. In these cases, the specializer must make reduce/residualize choices for expressions which depend on the return value of a residual conditional expression or procedure call. We found that performing equality testing on values at specialization time, and basing reduce/residualize decisions on such tests allowed an online specializer to build less general return values and perform more reductions in code depending on those values.

In a naive polyvariant specializer, merging of approximations of downward-passed values (formal parameters) never happens: a call site may re-use an existing specialization only when all of the known values in its arguments match those in the argument vector used to construct the specialization. Otherwise, a new specialization is built. Maximal preservation of common information occurs vacuously, since different argument vectors are never merged.

As we saw in Section 2.2, a real specializer may have to build a single specialization which is applicable at multiple call sites which have different argument vectors. First, the specializer will only terminate on a loop if the initial and recursive entry points of the loop call the same specialization. Second, a specialization of a first-class function must be applicable at all call sites which might be reached by that specialization at runtime.

In the first case, the specializer must build a single specialization which is applicable from both the initial and recursive entry points of a loop. For instance, if we specialize the `length` program

```
(define (length l ans)
  (if (null? l)
      ans
      (length (cdr l) (+ 1 ans))))
```

on `l` unknown and `ans` known to be `0`, the specializer must build a specialization which is valid for all non-negative integral values of `ans`.

Offline specializers construct such sufficiently general specializations by deciding, at annotation time, which portions of a function's arguments to use when building specializations and comparing them in the cache; to ensure termination, the annotation phase must assure that the portions which are used will assume a finite number of values at specialization time. In this case, the specializer can use the type of `ans` when building the specialization, but not its value. The main problems are that proving finiteness may be difficult, and

that the finiteness of an argument might depend on specialization-time data; in both cases, failure to make use of argument values can lead to overly general specializations. This is less problematic than one might expect, because many offline specializers rely on the user to provide the generalization annotations. Users can solve the first problem by performing "proofs" themselves, and the second by being willing to make use of knowledge about specialization-time values, and making annotations that are valid only for these values. Often, this is discovered empirically; if the the specializer fails to terminate in a reasonable amount of time, the user goes back and adds some generalization annotations.

Online specializers accomplish this "generalization for termination" via a different means. When it decides to residualize the recursive call to `length`, the specializer compares the arguments to the recursive call with those to the initial call, and finds that they are different. Since the specialization must be valid for both sets, it computes a generalization of the arguments; in this case, "unknown" for `l` and "unknown non-negative integer" for `ans`. If the commonality had been data dependent, it would still have been discovered. Having concrete values which can be compared at specialization time gives online specializers the ability to detect properties which might be difficult, or even impossible, to prove before all of the specialization-time information is available.

Thus, we expect online specializers to retain more information about argument values in residual loops, particularly those in which the commonality between the initial and recursive calls is data dependent. Such data dependence occurs in the case of loops controlled by arguments to the program being specialized, a common feature of interpreters and simulators. For the sake of brevity, we will not give an example of such data dependence here.

The second case in which specializers must build a single specialization for multiple call sites (and thus compute a single approximation to the values of the specialization's parameters) is when a residual `lambda` expression can reach multiple call sites. Such cases occur when specializing higher-order programs, or interpreters for higher-order programs. A particularly common case arises when a program with non tail-recursive loops is CPS converted. Consider the CPS converted form of `length`:

```
(define (length l k)
  (if (null? l)
      (k 0)
      (length (cdr l)
              (lambda (ans)
                (k (+ 1 ans))))))
```

specialized on unknown `l` and `k=(lambda (temp) (+ temp 99))`. In this case, each invocation of `k` could either be invoking the initial continuation `(lambda (temp) ...)` or the recursive one `(lambda (ans) ...)`. Also, each continuation can be invoked at one of two sites (`(k 0)` or `(k (+ 1 ans))`), and each specialized continuation must be valid for both call sites.

Unlike the case of upwardly returned values, in which the need to compute a single, general return value can be avoided by duplicating context, no amount of context duplication is sufficient to eliminate the need to compute a single, general specialization of each continuation in this case. A typical offline annotation phase will deduce that both the type and value of the parameters `ans` and `temp` will be known at specialization time, but will be unable to deduce that the values will vary between call sites while the types remain the same. Unless the annotation phase can prove that the types will remain the same, an offline specializer will build overly general specializations which don't take advantage of the type information. In the case above, the proof is fairly simple; a sufficiently smart annotation pass could indeed perform the proof, and generate the desired specialization, although, to date, no such annotation pass has been implemented.[18] Proofs can become more difficult; proving that all residual continuations in an interpreter are invoked on a store of identical shape requires several inductive steps.

In the worst case, a proof is impossible because the commonality between call sites is dependent on specialization-time data. The `fact` example on page 32 is such an example; as another simple example, consider a version of the `length` function in which the base case is not specified until specialization time:

```
(define (length l k)
  (if (null? l)
      (k base)                ;; base is free in length
      (length2 (cdr l)
               (lambda (ans)
                 (k (+ 1 ans))))))
```

If we specialize the program with `base=0`, the specializations of the initial and recursive continuations can use the fact that their parameters are integers; if we specialize it with `base=1.5`, then the specializations may only rely on their parameters being numbers. Unfortunately for offline specializers, both values for `base` have identical abstractions: "static"

[18] Such an annotation pass would have to make use of concrete values in the program, with all of the attendant complications.

value with "static" type, which isn't enough information to perform the requisite equality comparison (because we can't tell if the types will be equal). This example is deliberately trivial; such data dependent commonality does arise in the context of polymorphic programs, programs with error values, and interpreters for such programs.

Thus, we see that, in an offline framework, the utility of the CPS transformation is restricted to programs where all continuation applications are beta-reducible at specialization time; otherwise, the specializer would have to build a specialized version of the continuation which is sufficiently general to be applicable at multiple call sites. In other words, the CPS transformation is of use only when the residual version of the (direct style) original program is tail-recursive.

Consider the interpreter of Figure 2.2. Specializing the CPS-transformed version of this interpreter on a program with `while` loops but without procedure calls is not problematic; only the procedure `eval-while` will be residualized, and it is tail recursive, meaning that its continuation will be beta-reduced at specialization time. Any store accesses in that continuation will be able to take advantage of the shape of the store. Specializing the CPS-transformed interpreter on a program with recursive procedure calls is different. In this case, the procedure `eval-stmt` will be residualized, and its specialization will contain a non tail-recursive call to `eval-stmt`, due to the unfolding of `eval-begin`. The residual program might contain code like

```
(letrec
    ((eval-stmt49 (lambda (k store)
                    (if ...
                        (eval-stmt49 (lambda (store50) (k (... store50 ...))))
                        (k store)))))
  (eval-stmt49 (lambda (store51) ...) store49))
```

Unlike the tail-recursive case, in this fragment, the initial continuation to the loop, `(lambda (store51) ...)`, cannot be unfolded because both of its call sites are also reached by (closures constructed from) `(lambda (store50) ...)`. Unless the annotation phase can prove that both invocations of each continuation pass stores of identical shape, it will be forced to annotate all store accesses in both continuations as residual. In this case, the commonality between `store` and `store50` (they contain the same names, in the same order) is a function of the MP interpreter, and is thus (theoretically, anyway) deducible from the interpreter text alone. Not only might this be difficult for an annotation phase to prove, but it neglects any data-dependent commonality between `store` and `store50`; for instance, the

value bound to the identifier `x` might be the same, or have the same type, in both stores.

An online specializer can handle both the `length` and the interpreter examples correctly because by iteratively computing generalized values for the continuation parameters `temp` and `ans` at specialization time (an algorithm for doing this is given in Chapter 5 and in [97]). In cases where an offline system (or its user) would have to construct a difficult proof, an online system can provide equivalent output with significantly less effort, and in cases where data dependence is present, it can achieve results which are not possible under the offline method.

2.5 Related Work

This section describes work, in both the offline and online areas, on improving the accuracy of program specialization.

2.5.1 Offline

Offline specialization and binding time analysis were first developed by the Mix [64] project at DIKU, as a means of achieving self application. Since then, the accuracy of offline specialization has steadily increased, due to the development of both more accurate binding time analyses and various program transformation strategies called "binding time improvements."

Mogensen [83] developed a binding time analysis that handled structured data; Consel [22] developed a polyvariant version. The binding time analyses for higher-order languages due to Bondorf [11] and Consel [23] are monovariant, while those of Mogensen [83] and Rytz and Gengler [98] are polyvariant. More recent work on *facet analysis* by Consel and Khoo [29] has further increased the accuracy of binding time analysis by allowing it to make use of known properties of unknown values. None of this work, however, solves the fundamental problems with annotations themselves: certain information is simply not available until specialization time, and cannot be made available at annotation time.

Binding time improvement via the replication of code is common practice in writing interpreters to be specialized via offline means (for some example interpreters written in this style, see [12]). Automating such replication was first suggested by Mogensen [83]. Using the CPS transformation to perform the same task was first suggested by Consel and Danvy [26] and has been used to improve binding times in several examples [24, 71].

Bondorf's "CPS-less" specializer [13] provides some binding time improvements without requiring CPS-transformation of the input program. Other manual transformations, such as eta-abstraction, are also common; Holst and Hughes [55] suggest a possible means of automating such transformations.

2.5.2 Online

The early program specializers of Kahn [67] and Haraldsson [49] were, to some degree, online specializers. Turchin's supercompiler [111], which can be considered a superset of polyvariant specialization, performs online termination analysis, folding, and generalization. Recent work on online specializers for functional languages includes the MIT scientific computing project [10, 8, 9], the Stanford FUSE project [115], and others [46, 101, 43, 80]. Online methods are also popular in the logic programming community [100, 107].

Recent accuracy gains made in online specialization have been due mainly to specializers' ability to represent more detailed information about runtime values (the *partials* of [101], the *placeholders* of [10], the *symbolics* of [115, 116] and the *facets* of [29]), their willingness to use this information in a first-class manner (early specializers such as [49] severely limited the use of type information in data structures), and the use of generalization and fixpointing techniques to retain more information across residual calls [116, 97, 100].

Other useful advances in online specialization have dealt with representational issues, such as preventing code duplication without compromising accuracy by delaying some reduce/residualize decisions until after specialization [114, 115], and limiting the construction of redundant specializations through the use of type information [92, 93]. Termination mechanisms are also an active area of research; for some recent strategies, see [115, 100].

2.6 Summary

We have described the offline and online approaches to program specialization in some detail, and have shown several instances in which the use of statically-generated annotations can compromise accuracy in spite of the use of binding time improvement techniques. In general,

these accuracy losses are due to either

- The need to represent multiple specialization-time contexts via a single annotation, or
- The inability to take advantage of commonality between contexts without making reduce/ residualize decisions at specialization time.

We have shown that, in some cases, further accuracy losses are due to the limitations of present-day annotation techniques, rather than the use of offline strategies *per se*.

In all of our examples, it has been the case that online specializers provide equal or better accuracy with less need for "pre-transformation" of the input program. We believe this represents useful progress toward an eventual goal of building good specializations of arbitrary user programs with minimal user intervention. The remainder of this dissertation addresses the implementation of an online specializer, describing several algorithms for improving the quality of residual programs through more careful maintenance and use of information at specialization time.

This increased accuracy comes at a cost in performance; it is our hope that some of the techniques used to provide efficient offline specialization will be transferable to the online world. We also expect that many of the techniques used by binding time analyses to reason about specialization-time data structures at annotation time may be useful for reasoning about runtime data structures at specialization time, resulting in even more accurate specializations. Conversely, the offline methodology might be improved by using online mechanisms to utilize concrete values in the program at annotation time, or to make some termination-related decisions at specialization time.

Chapter 3

The Structure of an Online Program Specializer

This chapter describes the structure and operation of a basic online specializer, in preparation for later chapters addressing more specific techniques. We will describe a specializer, FUSE, for a functional subset of Scheme with pairs (`cons` cells) and higher-order functions, but no strings, vectors, input/output or `call-with-current-continuation`.

The FUSE project developed a number of program specializers (all named "FUSE" in the literature) with varying capabilities. These specializers share certain basic characteristics but were extended in different, sometimes incompatible, ways. The specializer described in this chapter does not correspond exactly to any extant version of FUSE, but rather to those "base" characteristics common to all versions of FUSE.

We begin with an overview of FUSE's structure as seen from the outside, and describe the input language, argument specifications, and output language. Section 3.2 treats data structures internal to the specializer, concentrating on the use of *symbolic values*. In Section 3.3, we describe FUSE's algorithms for constructing specialized code. We conclude with a small example.

3.1 Overview

Like many program transformation systems, FUSE is structured as several phases (*c.f.* Figure 3.1). A *front end* translates a program written in the input language (a subset of Scheme) into a restricted kernel language. This program is then passed, along with a

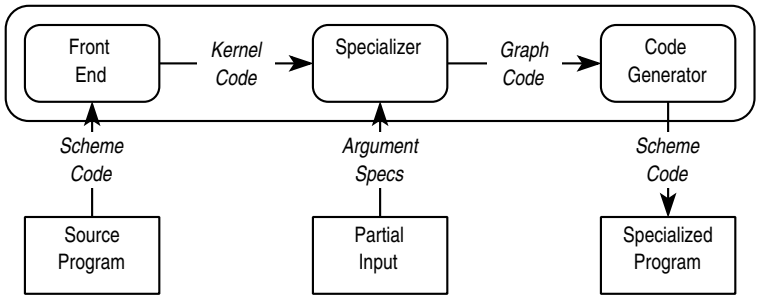


Figure 3.1: Basic structure of the FUSE program specializer.

specification of the inputs on which it is to be specialized, to the *specialization* phase, which produces as output a residual program expressed as a dataflow graph. A *code generation* phase converts the program graph back into the input language (or, optionally, into a different language).

The front end and code generation phases contain little interesting technology and are almost identical to similar phases used in compilers. We will not describe them separately, but will instead treat them when we describe the input and output languages used by the specializer. The remainder of this section describes the structure of the specialization phase's inputs and outputs, while subsequent sections describe the specializer's internal data structures and algorithms. The specializer has two inputs: a source program written in the kernel language, and a description of the values on which it is to be specialized, and one output: the residual program, expressed as a dataflow graph.

3.1.1 Input Language

FUSE specializes programs written in a functional subset of Scheme. Because many Scheme syntactic forms can be expressed as the composition of only a few special forms, the front end translates programs in the Scheme subset into a kernel subset which has very few special forms. This macro-expansion into a kernel language is encouraged by the language definition [88], which separates syntactic forms into *essential* and *derived* forms, and is common practice in Scheme compilers and program transformation systems. We expect that the reader is familiar with this translation process, and will usually couch our examples in terms of Scheme source programs rather than kernel language programs.

```
k       ::= constants (numbers, symbols, booleans, ...)
v       ::= variable names
prim    ::= primitive names
fcn     ::= function names
exp     ::= (CONST k) |
            (VAR v) |
            (IF exp exp exp) |
            (LET (v exp)* exp) |
            (PRIMCALL prim exp*) |
            (USERCALL fcn exp*) |
            (LAMBDA v* exp) |
            (ABSCALL exp exp*)

program ::= (LETREC (fcn (LAMBDA v* exp))* fcn)
```

Figure 3.2: The input language processed by the specialization phase

Expressions in the kernel language consist of 6-8 special forms (depending on the version of FUSE): constants, variables, conditionals, `let` bindings, primitive applications, and user function applications (higher-order versions of FUSE add `lambda` expressions and applications where the function position is evaluated).[1] A program consists of a set of function definitions binding names to procedure definitions, and the name of the "goal" procedure (the one to be specialized). This is accomplished using a single `letrec` expression at top level. A syntactic description of the kernel language is given in Figure 3.2.

Unlike some Scheme compilers, we treat `let` primitively. This is not necessary, since all variable binding can be expressed via function calling, but it is easy to implement and its use speeds up specialization significantly: (1) unlike function calls, all `let` expressions are unfoldable,[2] and (2) handling `let` primitively in the specializer is much faster than constructing and applying closures.

Note also that this syntax restricts named recursion to the top-level `letrec` expression;

[1] For the sake of simplicity, and without loss of generality, we are writing as though all versions of FUSE syntactically distinguish primitive, user, and abstract calls. Some versions do this (for the sake of convenience, the front end automatically determines the type of each procedure call in the source program), while others have only one syntactic form, `CALL`, and determine the types of call heads at specialization time. Because our subset of Scheme is functional, we use the terms "function" and "procedure" as synonyms.

[2] All `let` expressions are unfoldable because (1) our language has no side effects which can be duplicated or lost due to beta-substitution, and (2) FUSE's graph-structured output language guarantees that no computation duplication is introduced as a result of substitution (*c.f.* Section 3.1.3).

embedded `letrec`s are not permitted. This restriction simplifies the implementation (and description) of the specializer and is present in several other program specializers [21, 14]. It has not been found to be overly restrictive. Some versions of FUSE lift this restriction by permitting embedded `letrec` expressions, while others eliminate embedded `letrec`s using a simple preprocessing phase similar to *lambda-lifting* [60, 87]. The function names bound by the top-level `letrec` may be used only in the function position of `USERCALL` forms and may not appear in any other context. We call such functions "top-level functions" and their call sites "top-level function calls."[3] Such functions may well be higher-order (*e.g.,* they may accept functions as arguments), but may not be passed as arguments to higher-order functions.

Two additional restrictions are not captured by the grammar of Figure 3.2. First, the function head of a `ABSCALL` expression must evaluate to a closure (the result of evaluating a `lambda` expression)—primitives and functions bound by the top-level `letrec` may not be used in a first-class manner. This is easily solved by eta-abstraction; that is, converting `(foo +)` to `(foo (lambda (a b) (+ a b)))`; although this is not automatically performed by the front end, it easily could be. We call the functions created by `LAMBDA` forms and invoked by `ABSCALL` forms "closures" or "first-class functions," and call their call sites "closure calls" or "first-class calls." Second, all recursions must be broken by a call to a top-level function; this is required because the termination mechanism only guarantees against infinite unfolding/specialization of top-level functions, and code using the Y operator or streams might result in nontermination of the specializer. Once again, this restriction need not be made on source programs; the same preprocessing phase which performs lambda-lifting also factors each `lambda` body into a top-level function parameterized by the body's free variables.

Finally, some versions of FUSE add termination-related annotations to the syntax of the kernel language. Although we will discuss these briefly in our treatment of termination mechanisms (*c.f.* Section 3.3.4), we will omit them from all other text. The methods described in this dissertation are largely independent of particular termination methods,[4] so we will not clutter our descriptions or analyses with annotations specific to any particular mechanism.

[3] The names `USERCALL` and `ABSCALL` were chosen to coincide with that of previous work in partial evaluation [12, 22] that distinguished between "user-defined functions" and "user-defined abstractions."

[4] See Section A.1.5 for the properties we require of a termination mechanism.

$$
\begin{array}{lcl}
\mathit{Val} & = & \mathit{Scalar} + \mathit{Pair} + \top_{\mathit{Val}} \\
\mathit{Pair} & = & \mathit{Val} \times \mathit{Val} \\
\mathit{Scalar} & = & \text{Scheme scalar values (1, 2.5, foo, \#t, ...)}
\end{array}
$$

Figure 3.3: A very simple type system

3.1.2 Specifications

In addition to the program, the specializer also must be given a specification of the values on which the residual program may be invoked at runtime; it uses this information to perform reductions inside the body of the source program, resulting in a faster residual program. A specification is a finite description of a possibly infinite collection of values that might appear in its place at runtime. Because such a description is finite, it is necessarily approximate; to ensure correctness, it must also be conservative. That is, any specification must denote at least those values which will be passed as arguments at runtime; often it will denote more values.

The definition of a specification corresponds nicely with our notion of a *type*, since types are also used to describe sets of possible runtime values. Thus, like other specializers, FUSE represents possibly infinite sets of runtime values using approximations drawn from a type system. Many type systems could be used for this purpose; we have used several in FUSE (which has a replaceable type system module). One very simple system is described by the domain *Val* of Figure 3.3, where scalars denote themselves, pairs denote all pairs whose `car` and `cdr` fields are denoted by the `car` and `cdr` of the specification, respectively, and $\top_{\mathit{Val}}$ denotes all values. This specification scheme represents completely known values exactly, and preserves the structure of partially known values with known, finite structure. It cannot, however, describe the structure of recursively structured objects of unknown size (such as the set of all lists of integers), disjoint unions (the set $\{1.2, 3, 5\}$), or arbitrary constraints (such as the set of all pairs of numbers whose sum is 3, or the set of values that are not pairs).[5]

At this point in our description, an element of this type system is useful only for specifying a single specialization-time value (*i.e.,* the argument to a unary function or primitive).

[5]Indeed, the `car` and `cdr` fields of a pair are specified completely independently; there is no way to specify any relationship between the two. Doing otherwise would require a constraint system, something which has not yet been implemented in any program specializer for functional languages.

We call such specifications *value specifications*. To specify all of the arguments to a multiple-arity function, we extend the type system in the obvious way: simply construct a tuple, the *argument specification*, containing the specifications for each argument. These tuples are interpreted similarly to the specifications for pairs; *i.e.,* the specification tuple denotes all argument tuples whose elements are denoted by the appropriate tuple elements. Later, when we define relations on specifications, they will also be extended to tuples in the obvious way. Because the tuple construction for specifying arguments is so simple, we will omit it from all type system descriptions, and simply assume that all type operations on value specifications also work on argument specifications.

Most versions of FUSE use extended forms of the simple type system above. One common extension is to extend the "flat" domain *Scalar* into a lattice with "top" elements for various subcollections of scalar values. For example, we might have

$$\begin{array}{lcl} \mathit{Scalar} & = & \mathit{Number} + \mathit{Symbol} + \mathit{Boolean} + \ldots \\ \mathit{Number} & = & \top_{\mathit{Number}} + 1 + 1.1 + \ldots \\ \mathit{Symbol} & = & \top_{\mathit{Symbol}} + \mathrm{a} + \mathrm{b} + \ldots \\ \mathit{Boolean} & = & \top_{\mathit{Boolean}} + \#\mathrm{t} + \#\mathrm{f} \\ \quad \vdots & & \end{array}$$

where $\top_{\mathit{Number}}$ specifies a runtime value known to be a number, etc. Another extension, used in higher-order versions of FUSE, adds closure values:

$$\begin{array}{lcl} \mathit{Val} & = & \mathit{Scalar} + \mathit{Pair} + \mathit{Closure} + \top_{\mathit{Val}} \\ \mathit{Pair} & = & \mathit{Val} \times \mathit{Val} \\ \mathit{Scalar} & = & \text{Scheme scalar values (1, 2.5, foo, \#t)} \\ \mathit{Closure} & = & \mathit{Exp} \times \mathit{Env} \\ \mathit{Exp} & = & \text{source code expressions} \\ \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Val} \end{array}$$

As we shall see, different choices of type system for value specifications don't affect our specialization algorithms, merely the implementation of primitives which use the specifications to perform reductions. For now, all that matters is that the choice of type system affects how precisely the user can specify the set of runtime values on which the residual program must operate.

3.1.3 Output Language

FUSE expresses residual programs in the form of dataflow graphs. Such graphs are very similar to expressions in the kernel language, except that they are not restricted to be

```
k         ::= constants
i         ::= variable names
prim      ::= primitive names
var       ::= (VAR v)
code-fcn  ::= (LAMBDA var* code)
code      ::= (CONST k) |
              var |
              (IF code code code) |
              (PRIMCALL prim code*) |
              (USERCALL code-fcn code*) |
              (ABSCALL code code*) |
              code-fcn
```

Figure 3.4: Dataflow graphs. Variables are factored out into a separate nonterminal so that we can restrict the formal parameters of `LAMBDA` nodes to be `VAR` nodes.

---

trees (*c.f.* Figure 3.4). That is, a single form may appear in multiple other forms. To help distinguish the graphical nature of these forms, we often call them *nodes* rather than *expressions*.

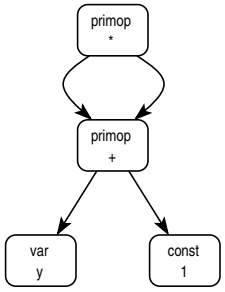
This sharing allows us to simplify binding constructs. First, there is no longer any need for the `let` construct; instead of binding a variable to an expression, then referencing the variable multiple times (as is done in source code), we simply construct multiple references to the expression itself. That is,

```
(let ((x (+ y 1)))
  (* x x))
```

is expressed as

`(* z z)` where[6] z refers to the expression `(+ y 1)`; *i.e.,*

[6] To describe graphs textually, we use a meta-notation: meta-variables like z refer to graph nodes, and "where" expressions bind these meta-variables to nodes.



Similarly, references to formal parameters in the body of a function are not variable references; the *same* variable node appears in the formals list and at all reference sites. Also, there is no need for a top-level `letrec` expression since call sites can directly reference function definitions (the only difference between a `USERCALL` and an `ABSCALL` being that the head of the `USERCALL` must be a `LAMBDA` expression; *i.e.,* the head position of a `USERCALL` is not evaluated).

There are two questions to be answered here. First, what advantage do we gain from producing such graphs? Second, how do we transform such a graph back into Scheme? We address each of these in turn.

Why Graphs?

As described in [114, 115], the primary motivation for using graphs is that they allow us to modularize the specialization process in a clean way by moving all issues of residual representation from the specializer into the code generator. This leads to a simpler, more efficient specializer, because many of the specializer's algorithms (pairwise generalization, iterative type analyses) repeatedly generate and discard fragments of residual code—it's more efficient to only worry about representational issues for code that will actually appear in the residual program.[7] Also, the code generator, which has access to the entire residual program, can often make better decisions (*i.e.,* produce better code) than the specializer, which at best has access only to a partially constructed residual program, and at worst considers only local information when making reduce/residualize decisions.

For example, FUSE uses graphs to avoid computation duplication. In conventional text-to-text specializers (*i.e.,* those which produce Scheme text without using an intermediate

[7] Moving work from the specializer into the code generator potentially hurts efficiency in self-applicable systems, because, typically, only the specialization phase is sped up by self application.

graphical representation), beta-reducing a function call (or `let` expression) substitutes the residual code expressions for the actual parameters (initializers) into the function's (`let` expression's) body. This can result in the duplication of some residual code expressions if the body references the formal parameters more than once. Such duplication can change a program's complexity, and (in the presence of side effects) its meaning. For example, consider

```
(let ((n (foo ...)))
  (+ n n))
```

where the expression `(foo ...)` is not reducible, so that `(foo ...)` is left as a residual expression. Naively unfolding the `let` expression would give residual code

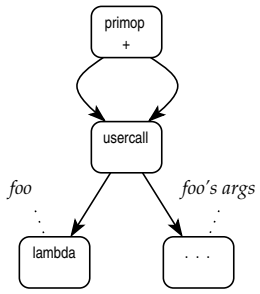
```
(+ (foo ...) (foo...))
```

which is undesirable because `foo` is applied twice as often in the residual program is it is in the source program. If the body of the `let` were instead `(+ n 1)`, we would want to unfold the `let` since it would save the construction of a binding for `n` and the execution of a reference to `n` at runtime.

In existing systems, this problem is addressed using static analyses which permit substitution only when the analysis can prove that no duplication can take place (this is done by permitting substitution only when the formal parameter appears at most once in each control path [14]), or that the cost of the duplication will be small [103]. Such approaches prohibit the specializer from performing (potentially beneficial) reductions if there is a risk of code duplication. Because the analyses are approximate, the specializer will sometimes fail to substitute expressions that could have been safely substituted. Although a post-pass can perform the substitution after specialization, it cannot, in general, perform all of the reductions the specializer could have performed had the substitution taken place at specialization time.

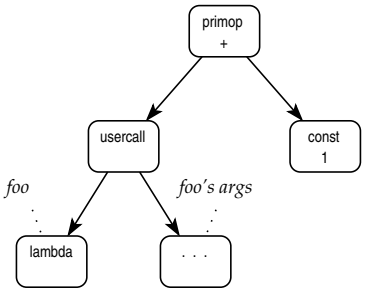
The use of graphs solves this problem because beta-substitution doesn't duplicate expressions; it merely adds references to an existing expression. Due to the evaluation of conditionals and simplification of primitives during the specialization process, references may come and go, but at code generation time, it is a simple matter to count the number of references to each expression in the graph and hoist it to a point dominating all of those references. Our first example above (the `(+ n n)` case) would produce the graph

`(+ x x)` where x refers to `(foo ...)`; *i.e.*,



Code generation would hoist the shared subexpression `(foo ...)` outside the `+`, yielding the desired (non-substituted) result. The variant with body `(+ n 1)` would produce the graph

`(+ (foo ...) 1)`; *i.e.*,



which requires no further hoisting. In this simple example, the static analysis and graph methods produce identical results. This is not always the case. Consider

```
(let ((n (foo ...)))
  (if x
      (+ 1 n)
      (+ n n)))
```

where `x` is known at specialization time, but `(foo ...)` is not. FUSE will generate the desired code regardless of the value of `x`, while static methods will prohibit substitution of `(foo ...)` even when `x` is true. The difference here is small, but in realistic programs it can be larger. For example, the runtime construction of an intermediate data structure can be avoided if all its accessors can be reduced at specialization time. Doing this requires opportunistically "substituting" the constructor so that the accessors can be reduced, and "hoisting" it back out if not all of them can be; static strategies for duplication avoidance tend to be overconservative in such cases. See [115] for more examples.

Translating Graphs to Scheme

Graphs differ from Scheme programs in two fundamental ways. First, the order of evaluation of the nodes in a graph is less constrained than the order of evaluation of the expressions in a Scheme program. Second, graphs contain no block structure; interprocedural sharing of bindings is expressed directly rather than with lexical scoping relationships. The code generator translates dataflow code into Scheme by removing these differences. We will treat each of these issues briefly; for a more detailed description, see Section 4 of [115]. The specialization techniques described in this dissertation are independent of the code generation algorithms used; this section is not intended as a guide to code generation algorithms (which were not part of this research), but merely as a starting point for the interested reader.

The order of evaluation of the nodes in a dataflow graph is constrained only by the dependency relationships between the nodes. That is, with the exception of `if` and `lambda` nodes, all nodes referenced by a node must be evaluated before that node can be evaluated.[8] This is not the same as the indeterminate order of evaluation of actual parameter expressions in Scheme, for two reasons: (1) in Scheme, the actual parameters must be evaluated in some serial ordering, while graphs allow parallel (interleaved) execution, and (2) because a graph node may have multiple fanout (*i.e.,* its output may be the input to more than one other node) even nodes in nonstrict positions (such as the consequent of a conditional expression with a false predicate) may still have to be evaluated if their output is used elsewhere.

Determining a correct serial order of evaluation would be fairly straightforward if our graphs were DAGs, as techniques for code generation from DAGs are well understood [2]. Our graphs are not DAGs because of recursion; the body of a `lambda` node may refer to the same `lambda` node. However, such circularities occur only with respect to function definitions: all "back edges" point only to `lambda` nodes. We break such circularities by introducing `letrec` expressions in the residual Scheme program, allowing us to proceed as though

[8] We are assuming that applications are strict in all of their arguments, as Scheme applications are. Of course, nothing prevents us from executing graphs in a lazy manner, or translating them into into "lazy" Scheme code using explicit delay and force operators. Indeed, some versions of the FUSE code generator do exactly that. Most do not, because the additional costs of memoization may make the residual program slower than the source program, a situation we would prefer to avoid. Similarly, we could easily execute graphs in parallel, or translate them into a language with explicit parallelism. Berlin's work on scientific computation [8, 10] makes good use of the parallelism exposed by the graphical output representation of his partial evaluator.

the graph were a DAG. In versions of FUSE where the source program is restricted to a single `letrec` expression at top level, this is particularly simple: collecting all specializations of top-level functions into a single `letrec` takes care of all circularities. Specializers which permit internal `letrec`s or which do not require that all recursions pass through top-level functions[9] can construct circular references to functions which capture variables, requiring a more complex strategy which can construct internal `letrec`s (see [115] for details).

Once the back edges have been dealt with, the code generator must determine the partial ordering induced by the remaining dependency relationships, then find a (potentially stronger) partial ordering which is expressible in Scheme. This can be done by naively generating Scheme code from the graph, while preserving sharing, then hoisting all multiply-referenced expressions to points which dominate all of their references.[10] Given that the back edges have been removed, none of these hoists will ever cross a procedure definition boundary (*i.e.,* nothing ever gets hoisted out of a `lambda` expression) because, in FUSE, functions are specialized only with respect to argument types (specifications), never with respect to argument expressions.[11] Residual code expressions are beta-substituted only when functions are unfolded, in which case there is no interprocedural sharing because the unfolded function has been incorporated into the specialization of the calling function.

Given that FUSE never duplicates the specializations of source expressions, but only constructs multiple references to the graph node representing the specialized code, there is always a valid ordering for the nodes which is expressible in Scheme, and which performs no more computation than the original program would have. The source program forces a (partial) ordering on the execution of source expressions, so simply forcing the residual program to execute the specializations of the source expressions in the same order which the corresponding source expressions would have been executed will yield a valid ordering. This argument says nothing about the difficulty of finding such an ordering, merely that such an ordering always exists; [115] gives one algorithm for finding an ordering. A valid ordering is not necessarily an optimal one, in that a hoisted expression may dominate

[9] The Y operator can construct circular environment dependencies which create a back edge from a function to itself. Forcing all recursions to pass through a top level function still allows the user to use the Y operator, but guarantees that circular references introduced by its use will be only to top-level functions, saving us the trouble of having to explicitly detect and handle circularities.

[10] Of course, only expressions whose execution requires computation need be hoisted. It doesn't make any sense to hoist Scheme expressions corresponding to constant or variable nodes.

[11] In particular, this means that when a function is specialized with respect to a closure, residual expressions in the closure's environment may not be beta-substituted into the function. Instead, the closure's free variables are added to the function's formal parameter list, as is done in [11].

control paths not containing any references to itself, causing the generated Scheme program to be more eager than necessary; validity means only that in such cases, the over-eager evaluation must have already been present in the source program. The code generator may (and often does) introduce additional laziness beyond this (*i.e.,* executing specializations of source expressions later than the source expressions would have been executed, or not at all), but it's never required to. The code generator is also free to perform additional hoisting (*e.g.,* common subexpression elimination and loop invariant hoisting), but the graphical representation is of little aid here, since expressions which were distinct in the source will be distinct in the graph—though traditional techniques such as those of [2] are still applicable on graphs.[12]

The other task of the code generator is to implement the interprocedural sharing of values in the graph using lexical scoping in the Scheme program. In the versions of FUSE described in this thesis, the only graph nodes that may be referenced by multiple functions are variable nodes; a formal parameter bound by one function may be referenced in the bodies of other functions. Given this restriction, computing the proper nesting of functions is simple: the Scheme `lambda` expression corresponding to any graph procedure (`lambda` node) p must be nested within all Scheme `lambda` expressions corresponding to any graph procedures q whose formal parameters are referenced by p's body. Such an ordering always exists because the graph procedures were constructed by specializing a lexically scoped source program—we can simply nest the procedure definitions in the order in which the closures of which they are specializations were constructed by the specializer. (Remember, the only procedures with circular references are specializations of top-level functions, which need never be nested. The body of each top-level graph procedure contains some finite number of `lambda` nodes, each of which can only possibly refer to variables bound by `lambda` nodes constructed before it was. Some versions of FUSE keep track of the lexical "parents" relation as specialization-time closures are constructed, and pass this to the code generator to aid in computing the nesting relation for the corresponding `lambda` nodes.)

[12] The use of graphs by the specializer allows the code generator to ensure that no residual expressions are duplicated as a result of beta-substitution; it does not guarantee against code bloat due to the specialization of duplicate expressions present in the source program.

3.2 Data Structures

Now that we have covered the external data structures of the specializer, we proceed to the internal data structures used during specialization. The specializer is similar to an interpreter, in that it executes the input program on data values, returning a data value. In this case, those data values are *symbolic values*, which represent specialization-time approximations to collections of runtime values, as well as the residual code necessary to compute those values. Like other interpreters, the specializer maintains an *environment* mapping identifiers to values (in this case symbolic values). Unlike an ordinary interpreter, though, the specializer cannot execute (unfold) all function applications—instead, *folding* behavior is achieved through the use of a *specialization cache*. This section describes these three data structures in greater detail.

3.2.1 Symbolic Values

Basics

FUSE uses data structures called *symbolic values* to represent specialized versions of source code expressions. When a source expression is specialized, the specializer returns a symbolic value with two attributes: a *residual code expression* which can be executed at runtime to obtain the same result as that obtained by executing the source expression, and a *value specification* which describes the set of values that might be returned by the residual code at runtime.[13] The specializer uses the value specification to perform reductions which depend on knowledge of runtime values. For example, if `(* (+ x y) 3)` is specialized on `x=1` and `y=2`, the symbolic value obtained by specializing `(+ x y)` will have a value specification of 3, allowing the multiplication to be reduced to 9. The residual code expression is used when source expressions cannot be fully reduced at specialization time. Consider `(* (+ x y) 3)` where `x=1` and `y=`$\top_{Val}$. In this case, the symbolic value s obtained by specializing `(+ x y)` will have a value specification of $\top_{Val}$ (or $\top_{Number}$, depending on the type system in use). This is insufficient information to allow the multiplication to be reduced, so it must be residualized. Construction of a residual code expression for the multiplication requires that we first obtain a residual code expression for the addition, which can be found in the residual code attribute of the symbolic value s (in this case, `(+ 1 y)`), allowing us to compute the

[13] Symbolic values can have several additional attributes, which will be introduced in later chapters.

final residual expression, `(* (+ 1 y) 3)`.

At first, it might seem as though the value specification and residual code expression attributes are never needed simultaneously, allowing symbolic values to be represented as *either* constant values *or* code expressions. Although this is often the case in other specializers, it is not the case in FUSE. There are several reasons. First, the value specification may give only partial information about the runtime values it denotes (*e.g.,* we know that a value is a number, but not *which* number, or we know that a value is a three-element list, but we don't know anything about the elements); this information is sufficient for performing some reductions (*e.g.,* reducing `number?` or `length`) but not all (*e.g.,* `+`). Thus, a residual code expression is always necessary, in case the symbolic value reaches an irreducible context. The partial information can sometimes be deduced from the code expression (*i.e.,* we know `(+ x 1)` returns a number, and `(cons a b)` returns a pair) but not always (*i.e.,* what does `x`, or `(foo (car x))` return?). Even in cases where the value specification is deducible from the code expression, it's often more efficient to keep it cached in the symbolic value instead of recomputing it every time we need to know it.

Second, the residual code expression can be useful even when the value specification is completely known, because it gives the code generator additional flexibility. For example, if `x` is bound to a symbolic value with value specification 1 and a residual code expression `x98`, while `y` is bound to a symbolic value with value specification $\top_{Val}$ and code expression `y99`, then the expression `x` is reducible (to `1`) while the expression `(+ x y)` is not. If the symbolic value bound to `x` contained only the value specification 1, then the residual code would necessarily be `(+ 1 y99)`. Retaining the code expression for `x` allows the code generator to choose between `(+ 1 y99)` and `(+ x98 y99)`, which, under some architectures (*c.f.* Section 6.4.3) may be just as efficient, yet lead to a smaller residual program. The idea here is that it's always valid to use a symbolic value's residual code expression in the residual program; the value specifications exist merely to allow for optimizations at specialization time.

Before we go on, we should note that both the value specification and residual code expression attributes of a symbolic value are immutable; that is, they are fixed at the time the symbolic value is constructed, and never change. This will not necessarily be the case for other attributes.

Specifications and Code

A symbolic value consists of a value specification and a residual code expression. We can express this as

$$Sval = Val' \times Code'$$

where Val' and $Code'$ are the value specification and residual code expression, respectively. As one might expect, these are very similar to *Val* and *Code* described above in Sections 3.1.2 and 3.1.3. One could indeed choose to have them be the same, but this complicates the implementation, making some optimizations more difficult to achieve. The problem is that in *Val*, the value specifications for structured values (*e.g.,* pairs and closures) are constructed solely from other value specifications. Thus, given a symbolic value denoting a pair, we can easily compute the value specification of the `car`, but cannot compute the symbolic value for the `car`. We can compute a symbolic value with equivalent attributes by consulting both the value specification (taking its first element) and the code expression (if it's an application of `cons`, use the first argument, otherwise wrap a call to `car` around it). Not only is this cumbersome, but each time we add an attribute to the symbolic value abstraction, we must find a way to compute the attributes of components of symbolic values denoting structures.

Our solution is to construct structured value specifications from symbolic values rather than from value specifications. For example, the type system *Val* of Figure 3.3, shown again here:

$$\begin{array}{lcl} Val & = & Scalar + Pair + \top_{Val} \\ Pair & = & Val \times Val \\ Scalar & = & \text{Scheme scalar values (1, 2.5, foo, \#t, ...)} \end{array}$$

induces the system

$$\begin{array}{lcl} Val' & = & Scalar + Pair + \top_{Val} \\ Pair & = & Sval \times Sval \\ Scalar & = & \text{Scheme scalar values (1, 2.5, foo, \#t, ...)} \end{array}$$

which is identical except for the references to *Sval* (rather than *Val*) in the definition of *Pair*. Since the value specification of a structured value actually *contains* the symbolic values of its components, we can easily perform copy propagation through intermediate data structures, and can preserve all attributes of symbolic values that are placed into and removed from

```
k         ::= constants
v         ::= variable names
prim      ::= primitive names
code-fcn  ::= (LAMBDA sval* sval)
Code      ::= NO-CODE |
              (VAR v)
              (IF sval sval sval) |
              (PRIMCALL prim sval*) |
              (USERCALL code-fcn sval*) |
              (ABSCALL sval sval*) |
              code-fcn
```

Figure 3.5: Code attributes of symbolic values

data structures. It should be clear that we can construct a system Val' of this nature for any type system *Val*, and that we can reduce elements of Val' to elements of *Val* via the appropriate projection (merely taking the value specification attributes of all components which are symbolic values). Thus, we can treat the value specifications of symbolic values as though they were elements of *Val*, and compare them using partial orderings on *Val*, even though they are really elements of Val'.

The code expressions in symbolic values are also a slight modification of the dataflow graph expressions *Code* of Figure 3.4. When generating Scheme code from the dataflow graphs, it is often useful to have the value specifications (and other symbolic value attributes) available along with the code expressions. For this reason, we use an altered set $Code'$ of dataflow graph expressions where recursive references to graph nodes (*Code* and *Var* in Figure 3.4) are replaced by references to symbolic values (*Sval*).[14] Also, since the code generator is now presented with symbolic values rather than graph nodes, there is no longer any need for nodes representing constants (*e.g.,* `CONST` code expressions) because the constant values can be obtained from the value specification attribute of the symbolic value. This allows the code generator greater freedom in generating code for constants, because it can now choose to generate either quoted values (taken from value specifications, as they would have been taken from `CONST` code expressions before), or applications of constructors (taken from residual code expressions). Because the specializer always builds residual code

[14]Lifting the restriction that formal parameters be variable nodes allows decisions on the inlining of constant values to be delayed until code generation time.

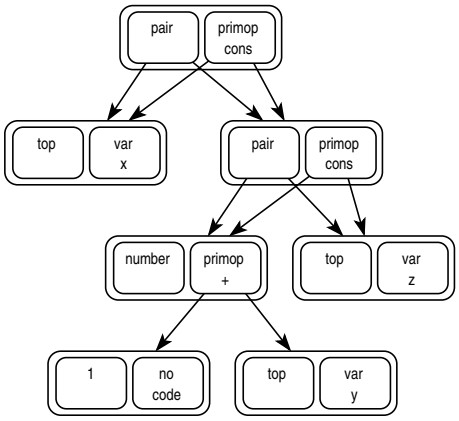


Figure 3.6: Symbolic value returned by specializing `(cons x (cons (+ 1 y) z))` on unknown `x`, `y`, and `z`.

expressions, it need no longer worry about what values are quotable in the code generator's output language—the code generator can itself decide whether to quote the values or leave the residual constructor invocations. Of course, some values (such as scalars) have no constructors, in which case the specializer uses `NO-CODE` as the code expression.[15] The altered dataflow graph language is shown in Figure 3.5.

As a simple example of symbolic values, consider specializing the expression

```
(cons x (cons (+ 1 y) z))
```

on `x`, `y`, and `z` unknown (*i.e.,* with value specifications of $\top_{Val}$). The resultant symbolic value is shown in Figure 3.6. Taking the value specification projection gives the value specification $(\top_{Val}\ \top_{Number}\ .\ \top_{Val})$, while taking the code expression projection gives the code `(cons x (cons (+ 1 y) z))`.

[15] FUSE generates residual code expressions for some, but not all, reductions producing scalars. In some cases (such as `(+ 1 1)` reducing to `2`), FUSE knows that the code generator will never make use of the residual code expression, and will optimize its space consumption by producing $\langle$ `2` , `no-code` $\rangle$. This is purely an optimization; the only requirement forced upon the specializer by the code generator is that every symbolic value that does not have a residual code expression must be quotable in the output language.

Instantiation

Clearly, from our discussion thus far, symbolic values have constructors and accessors, as well as projection functions for obtaining value specifications and residual code expressions. As it turns out, we need to define one more major operation involving symbolic values.

In FUSE, functions are specialized with respect to *argument specifications*, which are tuples of value specifications.[16] In other words, in computing the body of a specialization, the specializer is permitted access to the *types* of the actual parameters, but not to their *residual code expressions*, as this would yield inlining rather than specialization (*i.e.,* the specialization would be usable at only one call site). However, when the body of the specialization is being computed, the formal parameter names must be bound to symbolic values, not value specifications. This means we need a way to coerce value specifications into symbolic values. Similarly, when generalization takes place (*i.e.,* when the specializer must create a single symbolic value denoting all the values denoted by two other symbolic values, as is done when computing the symbolic value to return as the result of a specialization-time-undecidable `if` expression), the specializer first computes the general value specification by taking a least upper bound in *Val*, then constructs a symbolic value from that value specification. In both of these cases, the code expression is known: in the first case, it is the formal parameter, while in the second, it is the residual `if` expression.

We call this operation (of constructing a symbolic value from a value specification and a code expression) *instantiation*. If the value specifications used in symbolic values didn't contain references to symbolic values, instantiation would be simple: simply take the value specification and the code expression, and construct a tuple. Difficulties arise because every partially known or unknown component of the value specification must be given a corresponding code expression for computing the value at runtime. In the case of pairs and tuples, this means we must walk the spine of the value specification, generating accessor chains for each component. For example, consider specializing a function with formal parameter `a` on the actual parameter (a symbolic value) of Figure 3.6. We do this by taking the value specification of the actual parameter (in this case, $(\top_{Val}\ \top_{Number}\ .\ \top_{Val})$), and instantiating it on the formal parameter name (in this case, `(VAR a)`), obtaining the symbolic value shown in Figure 3.7. This symbolic value contains the same type information

[16] The implementation doesn't actually project out the value specifications into a separate tuple; it just ignores the irrelevant fields.

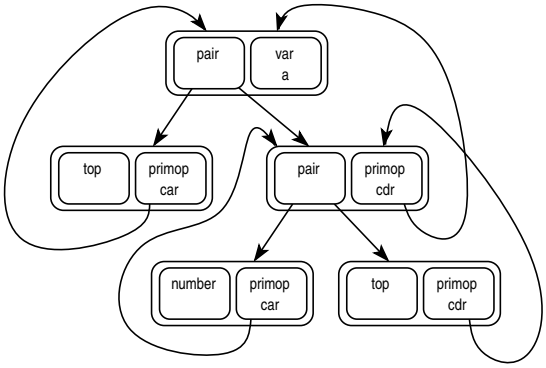


Figure 3.7: The symbolic value obtained by instantiating $(\top_{Val}\ \top_{Number}\ .\ \top_{Val})$ on `(VAR a)`

(*i.e.,* has the same value specification) as the actual parameter value, but all references to code expressions in the actual have been replaced by references to the formal.

Closures are more difficult because, in the absence of first-class environment operations, we cannot generate accessor chains (*i.e.,* there is no equivalent of `car` which can be applied to a closure to retrieve a value from its environment). Instead, for environment components with unknown values, we simply create a symbolic value using the value specification, and use a `VAR` expression generated from the corresponding free variable's name as the code (recursively instantiating if necessary), trusting that later arity raising will make sure that all free variables are bound before being used. "Upward" generalization out of an `if` expression is not handled by arity raising, so we simply forbid upward generalization of closures (given the existence of the CPS transformation, such generalization is unnecessary for higher-order programs). A sample implementation of the instantiation operation is given in Section A.1.4, which also defines some of its properties.

As an aside, we can easily implement *arity raising* simply by changing the instantiation procedure. Instead of constructing applications of accessor primitives around the formal parameter, we introduce a reference to a new formal parameter name for each unknown "leaf" of the actual parameter value. For example, under arity raising, the instantiated symbolic value of Figure 3.7 is replaced by that shown in Figure 3.8. This symbolic value is bound to the variable `a` at specialization time. References to the entire tuple (bound to `a`) now return applications of constructor functions over the new formal parameters

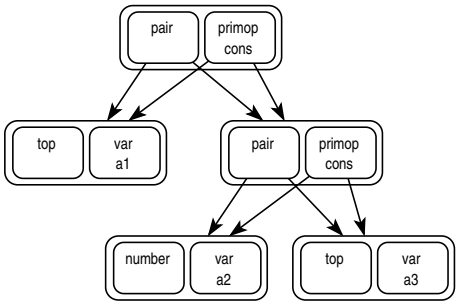


Figure 3.8: The symbolic value obtained by instantiating $(\top_{Val}\ \top_{Number}\ .\ \top_{Val})$ on `(VAR a)`, after arity raising.

representing `a`'s unknown components, while references to components of the tuple return the appropriate new formal parameter name.

3.2.2 Environments

Like other interpreters, FUSE maintains an environment that maps identifiers to values, in this case, to symbolic values. In versions of FUSE for first-order languages, the format of this environment is irrelevant; it is not a data object in the program being specialized, and thus need not be instantiated, generalized, compared, etc. When we consider higher-order programs, however, environments assume greater importance. Executing a `lambda` expression at specialization time creates a specialization-time closure object containing the text of the `lambda` expression and an environment mapping its free variables to symbolic values. When the closure is applied (unfolded or specialized), its environment is used to initialize the specializer's environment. Because closures are first-class data objects which can be passed anywhere, we must be able to instantiate, generalize, and compare environments.

An environment can be viewed as a (partial) mapping from Scheme identifiers to symbolic values, *e.g.*,[17]

$$Env = Var \rightarrow Sval$$

[17] Correct programs will have no unbound variable references, so the partial nature of the mapping is not an issue.

It may not be obvious how the (essentially structural) operations of instantiation, generalization, and comparison operate on environments. These operations are straightforward once we make the observation that only the *values* in the environment (*i.e.,* the "right hand sides" of bindings) will be manipulated by the residual program at runtime; the *identifiers* in the environment are immutable, specialization-time-only quantities. Because Scheme is statically scoped, none of the operations on an environment will change the "shape" of the environment, so we can treat environments as though they were mere tuples of values.[18]

Instantiation is simple: for each identifier bound in the environment, we examine the corresponding value slot, which yields a value specification. We then perform instantiation on this value specification and a code expression generated from the corresponding identifier.[19] The generalization and comparison operations know we will only ever generalize or compare environments with equivalent "shapes" (*i.e.,* environments from different closures of the same `lambda` expression). To compare two environments, we simply iterate over all identifiers bound by the environments, comparing the values to which they are bound in the two environments. Generalization is similar; we generalize values pairwise, then build a new environment mapping the old identifiers to the generalized values.

3.2.3 Specialization Cache

If FUSE were to operate only on completely known values, it would be able to unfold all procedure calls; all loops could be completely executed at specialization time. Actually, it's sufficient that the induction variable(s) be known; in some highly structured problem domains, this is indeed the case, and good results can be obtained without any construction or re-use of specializations [8, 10, 6].

FUSE, however, is a *program point specializer*; that is, it constructs specializations (essentially, parameterized blocks of residual code) of blocks of source code, and re-uses them at multiple points in the residual code (*i.e.,* builds multiple residual invocations of a single specialization). This allows for the construction of residual loops. In FUSE, the specializable program points are user function calls, and the specializations are expressed as function

[18] Indeed, we could rid ourselves of the identifiers entirely by performing environment conversion on the source program before specialization, replacing each variable reference with instructions specifying a chain of environment accessors which would then be executed on the current (implicit) environment.

[19] Simply put, the way to compute a closed-over variable's value at runtime is to look up that variable in the environment. As we mentioned above, this obligates the specializer to provide bindings for the closed-over variables at runtime, which it does by arity-raising the specialization(s) enclosing the unfolded closure body.

definitions in the residual program; parameterization is expressed through argument passing. Ignoring, for the moment, how FUSE decides when to construct specializations and which value specifications to construct them on (for details, see Section 3.3.4), the major issue in building specializations is deciding when it's necessary to build a new specialization, and when it's possible to simply build a residual invocation of an existing one.

The base version of FUSE is a typical *polyvariant specializer* [16], meaning that, for any given user function definition, it builds a different specialization for each different argument specification to which the function is applied. Thus, the construct/re-use decision can be made merely by determining whether a particular function has already been specialized on a particular argument specification. Like other specializers, FUSE uses a *specialization cache* to keep track of this information. The specialization cache is simply a (partial) mapping from function names and argument specifications to residual function definitions:

$$Cache = Fcn \rightarrow Val^n \rightarrow \textit{code-fcn}.$$

Given a function name and an argument specification (a tuple of value specifications, one per formal parameter), the cache either returns the distinguished token $\perp_{Cache}$, indicating that the specialization does not exist, or it returns a residual function definition, expressed as a *code-fcn*.[20]

Unlike the environment, the cache is not manipulated by the program being specialized, and does not appear in any "value" objects. Thus, operations such as instantiation and generalization are not defined on caches. We define caches here because they will be used when we reason about the specializer in later sections.

3.3 Algorithms

In this section, we describe how FUSE performs specialization for different source code constructs, and explain which portions of the specializer are responsible for various behaviors. The specializer can be viewed as a core module which handles Scheme special forms, and which is parameterized by four largely-orthogonal modules (*c.f.* Figure 3.9):

[20] A couple of technical points can be made here. First, the implementation maintains one cache per function name, removing one level of lookup. Second, because specializations may need to invoke themselves (to express recursion in residual programs), dummy cache entries containing *code-fcn*s with empty bodies are added when the specializer starts constructing a specialization, and the body field is updated once the specialization is complete. This distinction between "incomplete" and "complete" cache entries will cause minor difficulties in Section 6.3.3.

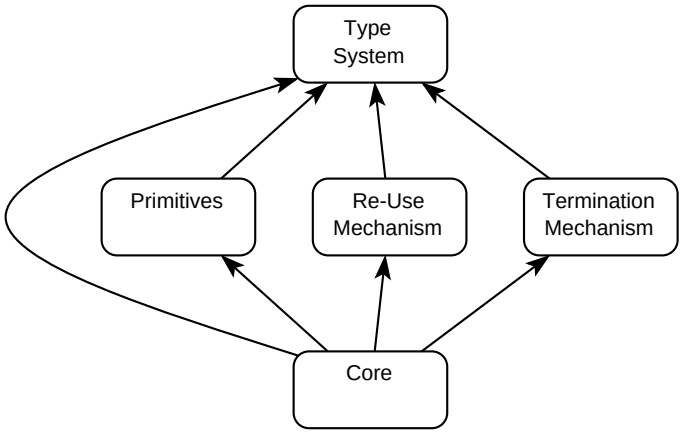


Figure 3.9: Calling relationships between modules in the specialization phase.

- **Type System**: This defines the specializer's repertoire of approximations to sets of runtime values (value specifications). A more complex type system will usually provide approximations to smaller sets of values, allowing more primitive operations to be reduced, and more polyvariance to be achieved. The type system defines value specifications using type lattices similar to Val' of Section 3.2.1, and also implements the instantiation, generalization, and comparison operations on symbolic values. We will not describe its internals further.

- **Primitive Operators**: These use the type information in their arguments to compute a return value, and to produce residual code if necessary. Primitive operators have both local consequences (deciding whether that particular operation will have to be executed at runtime) and nonlocal consequences (the return value may affect the reduction of other primitives at specialization time). We will describe several representative primitives in this section.

- **Specialization Re-Use Mechanism**: This decides when a previously-computed specialization of a procedure can be re-used at a particular residual call site, and when a new specialization must be computed. In base FUSE, this is very simple, involving only a lookup in the specialization cache (*c.f.* Section 3.2.3); Chapter 6 describes a more complex mechanism for this purpose.

- **Termination Mechanism**: At each procedure call site, this decides whether the specializer should unfold the call or build a residual call to a specialized procedure (the *unfold/specialize* decision). In the latter case, it also decides what portion of the information in the argument vector should be used in constructing the specialized procedure (the *generalization* decision). Not all termination mechanisms guarantee termination of the specializer; some speculative strategies use unsafe heuristics which can lead to nontermination. Although the methods in this dissertation are independent of the choice of termination mechanism used, for completeness, we will describe several in this section.

The remainder of this section is divided into five parts. The first describes specialization of special forms other than user function applications. We then describe the specialization of primitives and of function applications. The final two sections describe termination mechanisms and the handling of higher-order constructs.

3.3.1 Special Forms

The core module of the specializer looks very much like an interpreter; it performs syntactic dispatch based on the program argument, and maintains an environment binding names to symbolic values. Its internal state includes:

- the **expression** to be specialized,
- the **current environment**, so variable references in the expression can be evaluated,
- the **current cache**, so user function invocations in the expression can be specialized, and
- the **source program**, so the specializer can find the definitions of any user functions invoked by the expression, and
- **miscellaneous state** used by the termination heuristics, debugging support, etc.

The result of specializing an expression in some context is a symbolic value, and a (possibly) updated cache:

$$PE : Exp \rightarrow Env \rightarrow Cache \rightarrow Pgm \rightarrow Sval \times Cache$$

Some special forms are handled internally in the core module, while others are handed off to configurable modules. This section describes the handling of special forms internal to the core module (except for `lambda`, which we will defer to the discussion of higher-order constructs). A more formal description and associated correctness arguments appear in Section A.3.

Constants

The handling of constants is quite simple: FUSE merely translates the constant (a Scheme value) into a symbolic value. This consists of computing a value specification from the Scheme value and instantiating it on `no-code`.[21] Because the translation from Scheme values to value specifications depends on the type system being used, the specializer uses the type system to accomplish the translation.

Variables

Variables are the easiest special forms to handle. Because the environment maps variables to symbolic values (rather than just value specifications), FUSE looks up the variable in the environment and returns the resulting symbolic value; no instantiation is required.

`LET` expressions

Because FUSE expresses residual programs as dataflow graphs, `LET` expressions are always unfolded (*i.e.,* because `LET` merely allows the body to build multiple references to specializations of the initializing expressions, rather than substituting multiple copies of specialized initializers, there is never any risk of duplication).

First, all of the initializers are specialized in the current environment, producing symbolic values. Then, the environment is extended to bind the variables bound by the `LET` to the corresponding symbolic values obtained from the initializers. Finally, the body is specialized in the new environment, and that symbolic value is returned as the result of the `LET`. This is simpler than text-based specializers, which must return a specialized `LET` expression (rather than just a specialized body) if a variable bound by the `LET` could be referenced multiple times along a single control path.

[21] This is safe because all quoted scalar values in the source language are assumed to be "quotable" in the output language. Residual code to construct structured values will still be created.

Conditional expressions

FUSE first specializes the predicate expression, which returns a symbolic value. If that symbolic value's value specification denotes only true values (this includes not only known true values such as `#t` and `456`, but also $\top_{Symbol}$ and other unknown, but provably non-false types), FUSE specializes the consequent expression and returns that symbolic value. Similarly, if the predicate's specification is false, the specialization of the alternative is returned.

In the case of an unknown predicate, the specializer has two tasks: it must construct a residual conditional expression which will have the same effect as the original conditional, and it must determine a conservative approximation to all values that could be returned by that residual conditional expression. The code expression is computed by specializing both arms of the conditional, then constructing an `if` node whose sub-expressions are the symbolic values of the conditional, consequent, and alternative. The value specification is determined by computing the generalization of the value specifications of both (specialized) arms of the conditional. These two values are then combined into a single symbolic value via instantiation.

Note that this is the first instance in our description of a *reduce/residualize* decision: the specializer dynamically decides whether to reduce the conditional to one of its arms, or to build a residual conditional, based on the value specification returned by specializing the predicate. This shows the FUSE is an online specializer; an offline mechanism would make its choice by consulting precomputed annotations, and would always make the same decision regardless of the result of specializing the predicate.

3.3.2 Primitives

FUSE is an applicative-order evaluator; for any application (primitive or user) it first evaluates the arguments, then performs an action which depends on the type of the application form. In the case of primitive applications, the symbolic values obtained by specializing the actual parameters are passed to the implementation of the relevant primitive.

The interface to the primitive module is as follows: given a primitive name and a tuple of symbolic values representing the primitive's arguments, the primitive evaluator returns a symbolic value representing the primitive's result. That is,

$$\mathit{PE\text{-}prim} : \mathit{Prim} \rightarrow \mathit{Sval}^* \rightarrow \mathit{Sval}$$

The mechanics of looking up primitive names are uninteresting; in this section, we will show implementations of particular primitives (*i.e.,* mappings from tuples of symbolic values to symbolic values).

FUSE implements a subset of the primitives defined in [88]. Each primitive uses the value specifications of its (already specialized) arguments to determine the value specification of its result, and to choose which, if any, residual primitive application to construct. If a residual application is constructed, its arguments are the specialization-time arguments, expressed as symbolic values.

Once again, we see the online nature of our specializer, in that not only does the local reduce/residualize decision depend on the specialization-time arguments to the primitive, but also, the result of that decision is communicated to the remainder of the specialization process via the specialization-time return value constructed by the primitive. It should be clear that this strategy achieves maximal polyvariance, because each specialization-time application of a primitive is treated completely separately from all other applications of that primitive.[22]

In the remainder of this section, we will briefly examine (idealized) implementations of three representative primitives: `*`, `cons`, and `car`. Similar descriptions (couched in a different formalism) appear in the Appendix.

The primitive *

In this section, we describe possible implementations of the primitive `*`. Not only is it typical of arithmetic primitives, but its implementation is also similar to those of other scalar primitives operating on symbols, characters, etc., and to those of type tests such as `number?` and `symbol?`. The implementation of scalar primitives is heavily dependent on the type system used for value specifications. We'll begin with a very simple lattice which distinguishes only between unknowns, pairs, and known scalars (*c.f.* Figure 3.3). If

[22] Contrast this with the offline approach, in which a single occurrence of `+` in the post-BTA program must be either reduced in all specialization-time contexts, or residualized in all specialization-time contexts, even if it is reducible under some, but not all, of the contexts.

```
(lambda (sv1 sv2)
  (let ((v1 (sval-value-spec sv1))
        (v2 (sval-value-spec sv2)))
     (cond ((and (scalar-type? v1)
                 (scalar-type? v2)
                 (number? (scalar-value v1))
                 (number? (scalar-value v2)))
            ;; both args are known numbers, so reduce
            (make-sval (make-scalar
                        (+ (scalar-value v1) (scalar-value v2)))
                       no-code))
           (else
            ;; at least one arg isn't a known number, so residualize
            (make-sval top (make-code-application *-primop sv1 sv2))))))
```

Figure 3.10: An implementation of the * primitive

both arguments to * are scalars, then the specializer can reduce the primitive. That is, if both arguments are known numbers, it performs the multiplication; otherwise, it deduces (at specialization time) that the primitive application would result in an error at runtime, and it constructs a residual application of the primitive so that the error can be detected at runtime.[23] If either argument is not a scalar, a residual multiplication is constructed[24] Scheme code for this implementation of * is shown in Figure 3.10.

This implementation can be extended in several ways. We can add elements like $\top_{Number}$ and $\top_{Integer}$ to the scalar type lattice. This allows us to generate simpler residual code with fewer error checks and type dispatches, and also lets us constrain more precisely the range of return values. That is, specializing `(* x y)` on `x` and `y` known to be numbers gives `(number* x y)` (where `number*` need not check to see that its arguments are numbers)

[23] Alternately, it could generate residual code to print an error message, but we didn't think it worth the effort to speed up error branches. It might seem preferable to execute the error (*i.e.,* print a message and halt) at specialization time (rather than runtime), but this would be incorrect. The specializer has proven that if control ever reaches this point at runtime, an error will occur, but it has not proven that control will actually reach this point. Thus, it must construct a residual `error` form. Error handling introduces other tricky issues (such as preserving errors across dead-code optimizations, preserving the ordering of errors, etc.), which arise because errors are side effects. FUSE sidesteps the issue by treating only correct programs. Any reduction which might lead to an error is simply residualized in its original form (*i.e.,* `(+ 1 'a)` specializes to `(+ 1 'a)` rather than `(error "not a number")`). Preservation and sequencing issues are ignored.

[24] Once again, if either argument was known to be pair or other non-numeric type, we would have the option of constructing a residual `error` form instead of a residual multiplication.

```
(lambda (sv1 sv2)
  (make-sval (make-pair sv1 sv1)
             (make-code-application cons-primop sv1 sv2)))
```

Figure 3.11: An implementation of the `cons` primitive

whose result (value specification) is also known to be a number, while specializing `(* 5 z)` for `z` known to be an integer gives `(integer* 5 z)` and an integer value specification. These can be viewed as "partial reductions," where a partial result is computed, and simpler residual code is left behind. The implementation is straightforward; we just add cases to the type dispatch (the `cond` form in Figure 3.10). Another class of extensions are algebraic simplifications, which make use of the identities $x * 1 = x$ and $x * 0 = 0$, provided that x is known to be a number (when x might be non-numeric, we can either generate a residual type check guarding an `error` form, or just leave the original expression residual, as is done in FUSE). [25] Their addition can make things more difficult in other parts of the specializer (*c.f.* Section 6.4.3).

The primitive `cons`

The `cons` primitive (and, indeed, any constructor primitive for structured values) is very simple, as its execution involves no reduce/residualize decision. Because `cons` is a non-touching [47] operation, it never examines its arguments, and so can be reduced on any arguments whatsoever. The returned value specification is simply the pair whose `car` is the first argument and whose `cdr` is the second argument. The returned code is a residual application of `cons` to the two argument symbolic values. In the terminology of [115], `cons` is "both reduced and left residual." A sample implementation is shown in Figure 3.11.

This differs from most other specializers, which differentiate between the construction of pairs containing only ground values (typically "reducing" the application of `cons`, returning a quoted value), and pairs containing unknown values (typically "residualizing" the `cons`,

[25] Note that the second identity (involving multiplication by 0) is difficult to implement in an offline system because a primitive with a dynamic input is being reduced to a static value. If the 0 is a constant in the program, the BTA may be able to deduce this; otherwise, the binding time of the expression `(* x y)` (where `y` evaluates to 0) will not be known until specialization time, requiring that the enclosing expression perform a specialization-time binding time test, which is not allowed under the offline framework.

```
(lambda (sv1)
  (let ((v1 (sval-value-spec sv1)))
    (cond ((pair-type? v1)
           ;; it's a pair, so just take the car
           (pair-car v1))
          (else
           ;; not necessarily a pair, so delay until runtime
           (make-sval top
                      (make-code-application car-primop sv1))))))
```

Figure 3.12: An implementation of the `car` primitive

returning a residual application of `cons` to its arguments). We view this distinction as unnecessary, since no specialization-time reductions depend on it (`car` and `cdr` work exactly the same on ground and non-ground pairs). While the code generator does care (it uses ground/nonground information to decide when it can build a quoted constant), the vast majority of pairs constructed during specialization will be completely consumed at specialization time, and will not appear in the residual program. Thus, FUSE avoids computing such a "ground/nonground bit" until code generation time.[26]

The primitive `car`

The primitive `car` is typical of accessor primitives. If the argument is known to be of the appropriate structured type, then the accessor is reducible, and we simply return the appropriate element of the structure. If the argument is known to be of an inappropriate type, we can either return a residual application of the accessor (as is done in FUSE), or a residual application of an error primitive. If the type of the argument is not known, then we construct a residual application of the accessor. Sample code is shown in Figure 3.12.

3.3.3 Function Applications

Perhaps the most interesting portion of any specializer is its treatment of calls to user-defined functions. In this section, we will described FUSE's treatment of top-level function applications (`ABSCALLS` are deferred to Section 3.3.5). As in any Scheme evaluator, the

[26] Though it adds complexity, distinguishing ground and nonground pair values can be useful when the specializer is itself specialized (*c.f.* Section 7.4.2).

arguments are evaluated (specialized) before the application is performed, yielding a vector of symbolic values, or *argument specification*. Now, at a `USERCALL`, the specializer *always* knows what function is being invoked, so there is always enough information to reduce (*e.g.,* unfold) the application. Of course, given the presence of recursion, unfolding all calls would be a bad idea, since it could lead to nontermination of the specializer. Thus, the specializer must sometimes choose to residualize (*e.g.,* specialize) function applications, but, *unlike all of the reduce/residualize decisions described thus far*, it cannot make this decision using only the local information present in the argument specification. We will defer the description of how this decision is made to Section 3.3.4; the remainder of this section treats the specializer's operations *after* that decision is made.

Argument evaluation, function lookup, and unfolding are performed by the core module. If the call is to be residualized, the construction of the specialized function body and the requisite interactions with the cache are controlled by a helper function (described more formally in the Appendix) which, given a function name, argument specification, and cache, returns a "code function" and a (possibly updated cache):

$$\mathit{PE\text{-}spec} : \mathit{Fcn} \rightarrow \mathit{Val}^* \rightarrow \mathit{Cache} \rightarrow \mathit{Pgm} \rightarrow \mathit{Code\text{-}fcn} \times \mathit{Cache}$$

Unfolding

Reducing a top-level function call is very similar to evaluating a `let` form. A new environment is constructed by binding the formal parameters of the called function to the corresponding symbolic values in the argument specification, then the body of the function is specialized in this environment, and the resulting symbolic value is returned. Just as in the case of `let` forms, this beta-substitution is safe with respect to duplication because of the graphical nature of the residual code.

Specialization

When the specializer decides to residualize a top-level function call, a number of operations take place. In addition to returning a "residualize" directive, the termination mechanism will provide a new, possibly more general, argument specification on which the specialization is to be performed (this is necessary to avoid nontermination due to the construction of an infinite number of specializations of the same function); we'll call this the *general argument specification*. The specializer looks up the general argument specification in the

specialization cache to see if a specialization has already been constructed; if so, it constructs a residual invocation of this specialization on the original argument specification, and returns a symbolic value consisting $\top_{Val}$ instantiated on the code for performing the application.[27]

If the desired specialization has not yet been constructed, FUSE must build it by specializing the body in a suitable environment. Because we will re-use the specialization at multiple call sites, the specialized body should depend only on the value specifications of the arguments, not on the code expression portions of any of the arguments (since those would be unlikely to be the same at multiple call sites). To achieve this, FUSE constructs a new argument specification by instantiating each value specification in the general argument specification on the corresponding formal parameter of the procedure to be specialized; this effectively introduces a level of indirection through the formal parameter names, which is exactly what we want. FUSE then unfolds the function on the new, instantiated argument specification, under a cache which maps the general argument specification to a "code function" object with a dummy body. Once the unfolding is complete, the resulting symbolic value (representing the body of the specialization) is side-effected into the "code function" object.[28] Since no part of the specializer ever examines the body, the temporary presence of a dummy value will cause no problems. A residual invocation of this code function is constructed as described above for the "cache hit" case.

3.3.4 Termination

Both unfolding and specialization are fairly straightforward operations whose behavior is independent of the particular mechanism used to choose between the two. Indeed, all of the algorithms to be presented in this dissertation are independent of the particular termination method used, so long as it obeys some simple constraints (described in Section A.1.5). For completeness, however, we discuss termination briefly.

[27] Like most other specializers, "base" FUSE doesn't prove anything about the values returned by calls to specialized functions, so it must assume that such a residual invocation could return any value. Chapter 4 describes an extension to FUSE which allows it to compute better approximations to such returned values.

[28] This two-step process is necessary because the body of the specialization may wish to invoke the *same* specialization. Failure to update the cache before constructing the body of the specialization causes non-termination in such cases (*i.e.,* the specializer will keep building the same specialization again and again, rather than re-using the one currently being constructed).

To terminate, the specializer must avoid two pitfalls:

1. **Infinite Unfolding**, and
2. **Infinite Specialization**.

This can be accomplished by making two decisions at each call site:[29]

1. **Unfold/Specialize**: Should the specializer unfold the call, or should it construct a residual call to the a specialized version of the function? (The goal is to avoid infinite unfolding.)
2. **Abstraction**: What specialization should be invoked/constructed? That is, what argument specification should be used to search the cache and/or construct the specialized function body? (The goal is to avoid constructing an infinite number of specializations.)

Decision 1 forces the specializer to consider whether executing a given call could lead to an infinite loop, which reduces to the halting problem. Thus, all mechanisms for making this decision will be heuristic in nature. Our particular mechanism for making Decision 1 allows us to make Decision 2 very easily.

When to Specialize

FUSE makes the unfold/specialize decision using a recursion detection mechanism, often coupled with induction heuristics or finiteness annotations. The basic idea is that only recursive calls are troublesome: all nonrecursive calls can always be unfolded. FUSE detects recursions by keeping a specialization-time *stack* which keeps track of active (*i.e.,* currently being unfolded) procedures, along with the argument specifications on which they were invoked. At each call site, FUSE checks the stack to see if that procedure is already active; if it isn't, then it can be unfolded.

Although this method is completely safe (if recursions aren't unfolded, all unfoldings must be finite), it is far too conservative. Many programs contain loops which are unfoldable

[29] Some discussions of termination issues describe only one decision, the choice between unfolding and specialization, and miss the need for abstraction decisions. Even in offline specializers, abstraction is necessary; it's just that it is often (but not always; consider the need for the `generalize` operator in [14]) performed implicitly as a result of approximate binding time analyses.

at specialization time (*e.g.,* recursive syntax dispatch in interpreters, or iteration over an array of known size in numerical computations); if these loops are not unfolded, polyvariance will be lost, and many possible optimizations will not be achieved. What we need is a method for deciding when to proceed with unfolding even in the presence of a recursive call. These heuristics vary widely between different versions of FUSE.

The simplest method, used in many of our experiments, relies on *finiteness annotations*, which are applied manually to the source program (though algorithms such as that of [54] could be used). These annotations are "promises" that particular formal parameters will assume only a finite number of value specifications during specialization, and are most commonly applied to parameters that are induction variables. We then define a new comparison operator on argument specifications which is just like the existing equality test, except that only finite (annotated) argument positions are actually compared; all unannotated positions are considered to be equal. The decision procedure then becomes: unfold a recursive call only when the argument specification of the call differs from all of the argument specifications in all active calls to the same procedure, under our new comparison operator. This method works quite well, though it has drawbacks: placing the annotations can be time-consuming, and polyvariance can be limited, since there is only one set of annotations per source procedure.

The automatic termination mechanism described in [115] uses two other heuristics. The first, *induction detection*, works by defining a well-founded partial ordering on argument specifications, and unfolding a recursive call only when the recursive call's argument vector is strictly smaller (in the partial ordering) than the prior call's argument vector. This detects and unfolds static induction, such as `cdr`-ing down a list of known length, or counting down from some number to zero (this only works if a "natural number" type is provided; otherwise, we wouldn't know that the induction is finite at specialization time). Of course, it misses many cases where the iteration space, though bounded, doesn't map monotonically to the specializer's partial ordering (consider a list-valued argument which shrinks by two pairs, grows by one, shrinks by two, etc). Without help from the programmer in the form of annotations, we cannot hope to detect all such inductions, but this approach works well for recursive-descent programs such as interpreters.

FUSE's other heuristic is the method of *conditional markers*. Similar in concept to the *dynamic conditionals* method used in [14, 12], this method relaxes the termination guarantee, tolerating nontermination of the specializer for programs containing what Katz [70]

calls *true divergences* and *hidden divergences* (basically, programs which either diverge for some valid inputs, or contain segments which, although never reached at for any valid inputs, can be reached by the specializer due to the partial nature of its inputs). The idea is that the termination mechanism need consider only *speculative* loops; that is, loops broken by a conditional which cannot be decided at specialization time. Each time the specializer specializes the arms of a conditional with an unknown test, it pushes a *conditional marker* object onto the call stack. When the specializer encounters a call, it examines only those argument specifications which were pushed onto the call stack *before* the most recent conditional marker. This heuristic is unsafe, and is quite sensitive to the placement of conditionals guarding loops (the conditional may lie around the recursive call, or inside one of the argument expressions to the recursive call); however, it has proven useful in practice.

How to Specialize

The methods described above only decide when a function invocation is to be residualized; they do not address the problem of how to choose the specialization to be invoked at that call site. This problem reduces to choosing the argument specification on which to search the specialization cache and, if necessary, construct a specialization.

All versions of FUSE use the same method, *pairwise generalization*, for this purpose (other online specializers, such as [100, 43], use similar methods). The motivation behind this approach is as follows. FUSE stops unfolding because it fears it will enter an infinite specialization-time loop. Instead, it defers the loop until runtime by constructing a residual loop. The specialized procedure representing the body of the loop must be sufficiently general to be applicable at both its initial entry point and any recursive call sites within the loop. The call stack provides an entry point (the prior call whose argument vector was insufficiently "different" from that of the current call), while the current call site forms one of (potentially many) recursive call sites. At the very least, the body must be sufficiently general to handle arguments from these two sites, so FUSE computes a single argument vector denoting all values that could be passed in at either site (by generalizing corresponding pairs value specifications), and use that to construct the specialization. Specialization proceeds with a new stack consisting of all calls above the initial call, plus the new (general) initial call.

If, during that process, other recursive call sites are found, one of two things will happen. Either the specialization will be sufficiently general to be applicable at those sites (in which

case we build more residual invocations), or it won't be (in which case we generalize once more and start over). This generalization/respecialization process will terminate because, in all FUSE type lattices, the number of elements above any particular argument specification is finite, even though the lattice itself is not of finite height.

Compared with the static argument abstraction methods used by offline systems and with the dynamic argument abstraction methods used in Schism [21], pairwise generalization preserves more information because it discards only that information which would make loop formation impossible. The price paid is one of specialization-time efficiency, because the specializer constructs each loop body at least twice: once while discovering the recursive call, and once again while constructing the body on the generalized argument vector (and, in some cases, while re-constructing the body on successively more general argument vectors). It can also result in larger residual programs because each residual loop is preceded by at least one unfolded version of its body; we call these unfoldings *preambles*. Different versions of FUSE use various methods for discarding such preambles, including (1) capturing a continuation at each call site, and throwing back to that continuation if the termination mechanism decides that a different unfold/specialize choice should have been made there, and (2) using side effects to annotate a frame on the call stack when it is involved in a generalization, informing the unfolding code to replace the unfolded body with a call to a particular specialized procedure. These approaches have different performance characteristics and implementation-related constraints which will not be discussed further.

3.3.5 Higher-Order Constructs

Thus far, our discussion has been limited to first-order Scheme programs. This section describes the handling of higher-order constructs in base FUSE. We describe the evaluation of the `LAMBDA` special form, the unfolding and specialization of the resultant closures, and the effects on the termination mechanism.

`LAMBDA` expressions

When the specializer evaluates a `LAMBDA` form, it constructs a symbolic value with a *closure*-valued value specification and a code expression of `no-code`. The value specification is a record containing an environment mapping all of the free variables of the `lambda`'s body to their values at the time the `LAMBDA` was evaluated, along with the source code for the body of the `LAMBDA` expression. As in ordinary Scheme interpreters, no computation is performed

when a closure is constructed; the body is evaluated only when the closure is unfolded or specialized.

Unfolding

FUSE unfolds all closure applications (`ABSCALL` forms) where the head can be reduced to a single closure. This is safe w.r.t. termination because of the requirement that all loops be broken by calls to top-level procedures.

The unfolding process is similar to that for `USERCALL`s, except that the closure's environment, rather than the empty environment, is extended with the formals/actuals bindings and used when evaluating the body. The result of the unfolding is a single symbolic value denoting the residual code (and specialization-time value approximation) for the specialized body.

Specialization

In a higher-order program, not all `ABSCALL` forms will be reducible, because sometimes the head expression will not evaluate to a single specialization-time closure. Such applications must be left residual, as must all `LAMBDA` forms reaching such applications. The former is simple (just construct a residual `ABSCALL` whose components are the specializations of all of the source `ABSCALL`'s subforms), while the latter is more complicated.

Like most other program specializers for higher-order Scheme [11, 23, 45], base FUSE does not perform control flow analysis, so it doesn't know which closures reach which residual applications, or which closures can "escape" from the program as part of its final return value. Instead, FUSE computes a simple approximation to the set of all closures which might conceivably reach a residual application ("appear in a residual code context," in the terminology of [11]), and specializes these on completely unknown argument specifications. Thus, the specializations, though perhaps overly general, are guaranteed to be applicable at any call sites they reach, even (for returned values) sites not contained in the residual program. Chapter 5 describes an extension to FUSE, based on control flow analysis, which computes better specializations of first-class procedures.

FUSE is thus faced with two tasks: (1) determining which closures appear in residual contexts, and (2) specializing them. We accomplish (1) by tracking generalizations: whenever a single value is used to represent multiple values, one of which is a closure, we must residualize the closure (because the general value might reach a `ABSCALL` or be returned by

the program at top level). Generalization occurs *explicitly* when computing return values of residual `IF` expressions, and *implicitly* when residual `USERCALL` and `ABSCALL` expressions are constructed. For a `USERCALL`, any closure appearing in the actual parameters which doesn't appear in the corresponding position of the formal parameters must be residualized, as must all closures in arguments to residual `ABSCALL`s.[30]

Task (2), specialization, is accomplished by specializing the closure on a completely unknown argument specification (*i.e.,* $(\top_{Val}\ \top_{Val}\ \ldots\ \top_{Val})$), producing a "code function" object (just like the specialization of top-level function). A separate cache[31] keeps track of which closures have been specialized, and what the specialized bodies are. The code generator consults this cache each time it encounters a symbolic value with a value specification which is a closure.

Termination

The addition of higher-order constructs complicates the termination mechanisms. Recursion detection is made more difficult because we are now faced with two kinds of specialization-time loops: self-recursion (unfolding a procedure calls itself, which, when unfolded, calls itself...) and recursion among multiple instances of the same `lambda` expression (specializing a closure constructs a *new* closure of the same `lambda` expression, which, when specialized, constructs another new closure..., which is common in stream code). What we need to do is to extend the stack so that when a closure is applied (unfolded/specialized), we consider not only those calls which were active just before the closure was *applied*, but also those which were active just before the closure was *created*. Researchers at DIKU [109] have also noted this problem. Katz [69] has designed a display-based mechanism for this purpose, but it has not been implemented. Extant versions of FUSE either naively append the creation-time and application-time stack (this works, but residualizes too often), or just use the creation-time stack (this is sufficient for CPS-converted versions of first-order code; restoring the stack to a continuation's creation-time state on continuation invocation mirrors the popping of stack entries upon procedure return in the corresponding first-order

[30] It might seem simpler to specialize each closure at the moment of its creation, just in case it might appear in a residual context. This has two pitfalls. First, the vast majority of closures (just like the vast majority of `cons` pairs) do not appear in residual contexts; they are constructed and applied at specialization time. Such behavior is typical both for "wrapper" procedures such as the function argument to `map`, and for most continuations in CPS code. Second, specializing all closures can lead to nontermination under some termination heuristics.

[31] Actually, a side-effectable slot in each closure.

program).

Higher-order constructs also complicate the induction heuristics: the method of conditional markers stops too often. Because CPS-converted code is tail recursive, the stack is never popped, meaning that *any* conditional with unknown test inside a loop body will appear to guard that loop. Thus, our experiments with CPS-converted code have relied primarily on the finiteness-annotation-based termination method.

3.4 Example

In this section, we offer a small example. The procedure `map1+` in Figure 3.13 takes a list of numbers and returns a new list, each of whose elements is the corresponding element of the original list, incremented by 1. It calls `make-incrementer` to construct an incrementing procedure (a closure which, when applied, returns its argument incremented by 1), then applies it to each element of the list using the higher-order procedure `map`. When we specialize `map1+` on the partially known list `(1` $\top_{Integer}$ `.` $\top_{Val}$`)`, we expect that the overhead of constructing and applying the first-class incrementing procedure will be eliminated, and that the loop in `map` will be unrolled on the first two elements of the list. The remainder of this section describes how the residual code, shown in Figure 3.14, is constructed.

3.4.1 Specializing `map1+` on `lst=(1` $\top_{Integer}$ `.` $\top_{Val}$`)`

The specialization process begins when FUSE is invoked on the program to be specialized (*i.e.,* the code in Figure 3.13, plus the information that `map1+` is the goal procedure), and the argument specification on which `map1+` is to be specialized, namely `(1` $\top_{Integer}$ `.` $\top_{Val}$`)`. FUSE constructs a cache entry mapping `map1+` and this argument specification to an initially empty *code-fcn* object, which we will call *spec1*. To compute the body of the specialization, FUSE instantiates the argument specification on the code for the formal parameter `lst`, yielding the symbolic value *lst*[32] shown in Figure 3.15, which is then bound to `lst` in the specialization-time environment. Next, the argument expressions to `map` are specialized. The call to `make-incrementer` is evaluated first, and, since no prior calls to `make-incrementer` are active, the specializer decides to unfold it. This binds `inc` to the

[32] We will use italicized names, in both the text and in figures, as a meta-notation for symbolic values; these names are not actually created or used during specialization.

```
(define (map1+ lst)
  (map (make-incrementer 1) lst))

(define (make-incrementer inc)
  (lambda (x) (+ inc x)))

(define (map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map f (cdr l)))))
```

Figure 3.13: A small example involving pairs and closures. The result of specializing the function `map1+` on the list `(1` $\top_{Integer}$ `.` $\top_{Val}$`)` is shown in Figure 3.14.

```
(define (res-map1+ lst)
  (let ((temp (cdr lst)))
    (cons 2 (cons (int+ 1 (car temp))
                  (res-map (cdr temp))))))

(define (res-map l)
  (if (null? l)
      '()
      (cons (+ 1 (car l))
            (res-map (cdr l)))))
```

Figure 3.14: Scheme residual program for `map1+` specialized on `(1` $\top_{Integer}$ `.` $\top_{Val}$`)`, generated from the graph of Figure 3.20. Completely known parameters (such as `f` in `res-map`) have been removed, but arity raising has not been performed (arity raising would factor `lst` into three separate parameters).

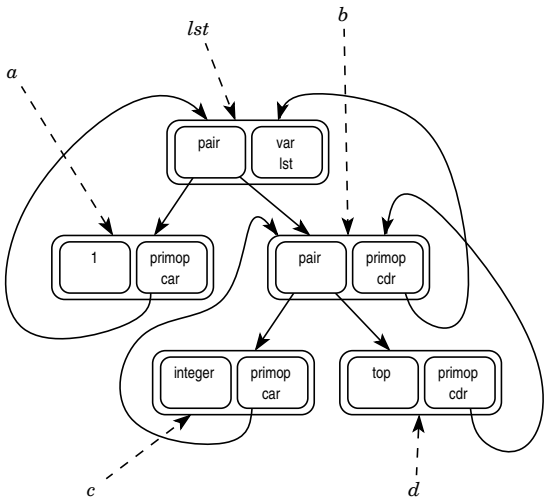


Figure 3.15: Symbolic value bound to `lst`

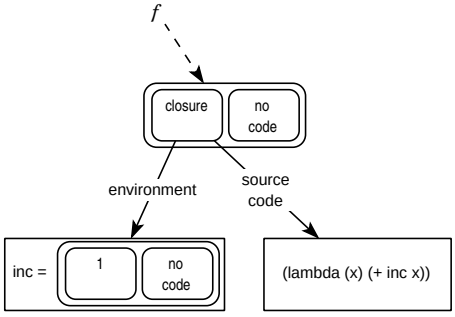


Figure 3.16: Symbolic value returned by unfolding `make-incrementer`

symbolic value ⟨ 1 , `no-code` ⟩, and executes the `lambda` form, which constructs the closure-valued symbolic value f depicted in Figure 3.16. The second argument, the variable `lst`, is dereferenced, yielding the symbolic value *lst*.

3.4.2 Unfolding `map` on `f=[closure ...]`, `l=`(1 $\top_{Integer}$. $\top_{Val}$)

When argument evaluation is complete, the procedure `map` is applied to these two symbolic values. The specializer decides to unfold, binding `f` and `l` to the closure- and list-valued symbolic values f and *lst*, respectively. Since `l` is nonempty, the primitive `null?` returns the symbolic value ⟨ *true* , `no-code` ⟩, and the alternative branch of the `if` is taken. Evaluation of the arguments to `cons` first evaluates `(car l)` (returning a), then `f` (returning f), and then applies `f` to ⟨ 1 , `(car` *lst*`)` ⟩ (returning ⟨ 2 , `no-code` ⟩). The second argument is itself an application, whose arguments evaluate to f and b, respectively.

3.4.3 Unfolding `map` on `f=[closure ...]`, `l=`($\top_{Integer}$. $\top_{Val}$)

The specializer detects that this call to `map` is recursive, because the initial call to `map` is still being evaluated. However, the termination heuristics decide to unfold the call anyway. This binds `f` to f and `l` to b, and evaluates the body again. Once again, `l` is nonempty, and the `if` takes the alternative branch. However, this time, `(car l)` returns a non-constant, namely c=⟨ $\top_{Integer}$, `(car` b`)` ⟩; when f is unfolded on this symbolic value, it returns ⟨ $\top_{Integer}$, `(int+` ⟨ 1 , `no-code` ⟩ c`)` ⟩. The expression `(cdr l)` yields symbolic value d=⟨ $\top_{Val}$, `(cdr` b`)` ⟩.

3.4.4 Unfolding `map` on `f=[closure ...]`, `l=`$\top_{Val}$

Yet again, the specializer is faced with a recursive call to `map`, which it once again chooses to unfold, evaluating the body of `map` in an environment where `f=`f and `l=`d. This time, `l`'s value is completely unknown (*i.e.*, d's value specification is $\top_{Val}$), so `(null? l)` returns ⟨ $\top_{Boolean}$, `(null?` d`)` ⟩ instead of a false value, so the specializer must evaluate both arms of the conditional. The consequent is a constant, ⟨ *nil* , `no-code` ⟩. Evaluating the alternative computes (car l), yielding e=⟨ $\top_{Val}$, `(car` d`)` ⟩ (*c.f.* Figure 3.19; when f is unfolded on this value, we get ⟨ $\top_{Val}$, `(+` a e`)` ⟩. The expression `(cdr l)` yields symbolic value g =⟨ $\top_{Val}$, `(cdr` d`)` ⟩.

At this point, the specializer sees a recursive call to `map` on an argument specification

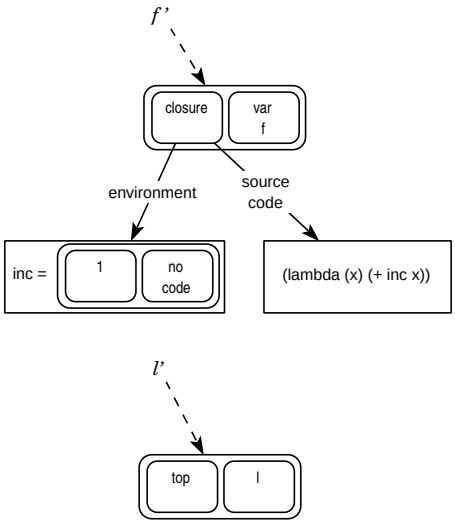


Figure 3.17: Symbolic values on which `map` is specialized

`f=[closure (lambda (x) (+ inc x)), inc=1], l=`$\top_{Val}$. The call stack looks like

```
map1+  lst=(1 ⊤Integer . ⊤Val)
map    f=[closure (lambda (x) (+ inc x)), inc=1],  l=(1 ⊤Integer . ⊤Val)
map    f=[closure (lambda (x) (+ inc x)), inc=1],  l=(⊤Integer . ⊤Val)
map    f=[closure (lambda (x) (+ inc x)), inc=1],  l=⊤Val
```

3.4.5 Specializing `map` on `f=[closure ...], l=`$\top_{Val}$

The termination heuristic decides that the current call and the most recent call (to `map`) have overly similar argument specifications (in fact, they are the same). Thus, it decides to build a residual call to the specialization of `map` on the generalization of the two argument specifications, which in this case is `f=[closure (lambda (x) (+ inc x)), inc=1], l=`$\top_{Val}$. Consulting the specialization cache, it finds that no such specialization exists, so it builds one. It adds a new entry to the cache, mapping the `map` and the general argument specification to a new *code-fcn* object, *spec2*. To construct the body of *spec2*, FUSE instantiates the general argument specification on the formal parameters of `map`, yielding the symbolic values f' and l' shown in Figure 3.17, then unfolds `map` on these values.

This unfolding proceeds exactly like the last one we described; when the recursive

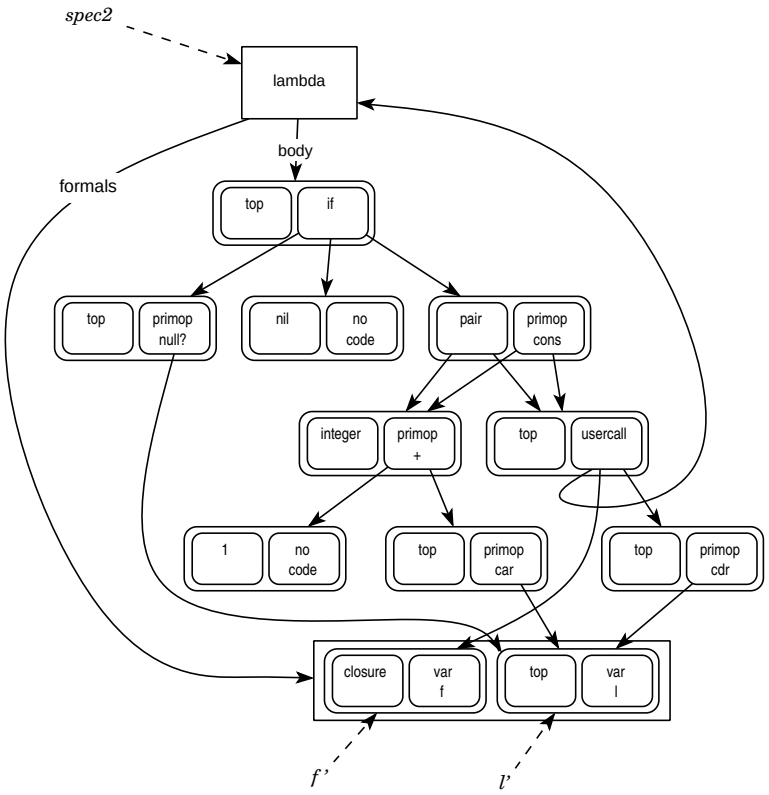


Figure 3.18: Result of specializing of `map` on `f=[closure ...], l=`$\top_{Val}$. We have abbreviated the value specification for the parameter `f`; see f' in Figure 3.17 for details.

call is reached, the termination heuristic once again decides to build a residual call, except that, this time, consulting the cache reveals that the desired specialization, *spec2*, already exists. Thus, the specialization of `(map f (cdr l))` returns the symbolic value $\langle \top_{Val}$, `(`*spec2 f′ l′*`)` $\rangle$[33]

Finally, both arguments to `cons` have been evaluated, allowing it to return a symbolic value, and in turn, allowing the `if` expression to return a symbolic value. This completes the construction of the body of *spec2*; the resultant "code function" object is shown in Figure 3.18.

3.4.6 Completing the unfolding of `map` on `f=[closure ...]`, `l=`$\top_{Val}$

Now that *spec2* is complete, the first residual call to the specialization can be completed, which in turn allows another pending `cons` and `if` to execute, giving the symbolic value of Figure 3.19. At this point, the specializer realizes it is faced with a *preamble*; that is, the symbolic value of Figure 3.19 denotes a single unrolling of *spec2*, followed by an invocation of *spec2*. Because this leads to a 50% growth in code size while eliminating only one procedure call, the specializer decides to eliminate the preamble by replacing it with an appropriate invocation of *spec2*.[34]

3.4.7 Completing the unfolding of `map` on `f=[closure ...]`, `l=(`$\top_{Integer}$ `.` $\top_{Val}$`)` and on `f=[closure ...]`, `l=(1` $\top_{Integer}$ `.` $\top_{Val}$`)`

The replacement of the preamble by a residual call finishes the unfolding of `map`, and the specializer returns to the body of `map1+`. Two more pending executions of `cons` are run, at which point the construction of the specialized function *spec1* is complete. The final result of the specialization process is shown in Figure 3.20; executing the code generator yields the Scheme code of Figure 3.14.

[33] Remember that, in base FUSE, all residual procedure calls are assumed to be capable of returning any value, thus the specification $\top_{Val}$ for the returned value.

[34] In this example, the specializer actually had to compute the preamble, then splice it out. Some versions of FUSE avoid this extra work by "rolling back" the computation to the call site which caused the preamble as soon as the decision to construct the specialization has been made.

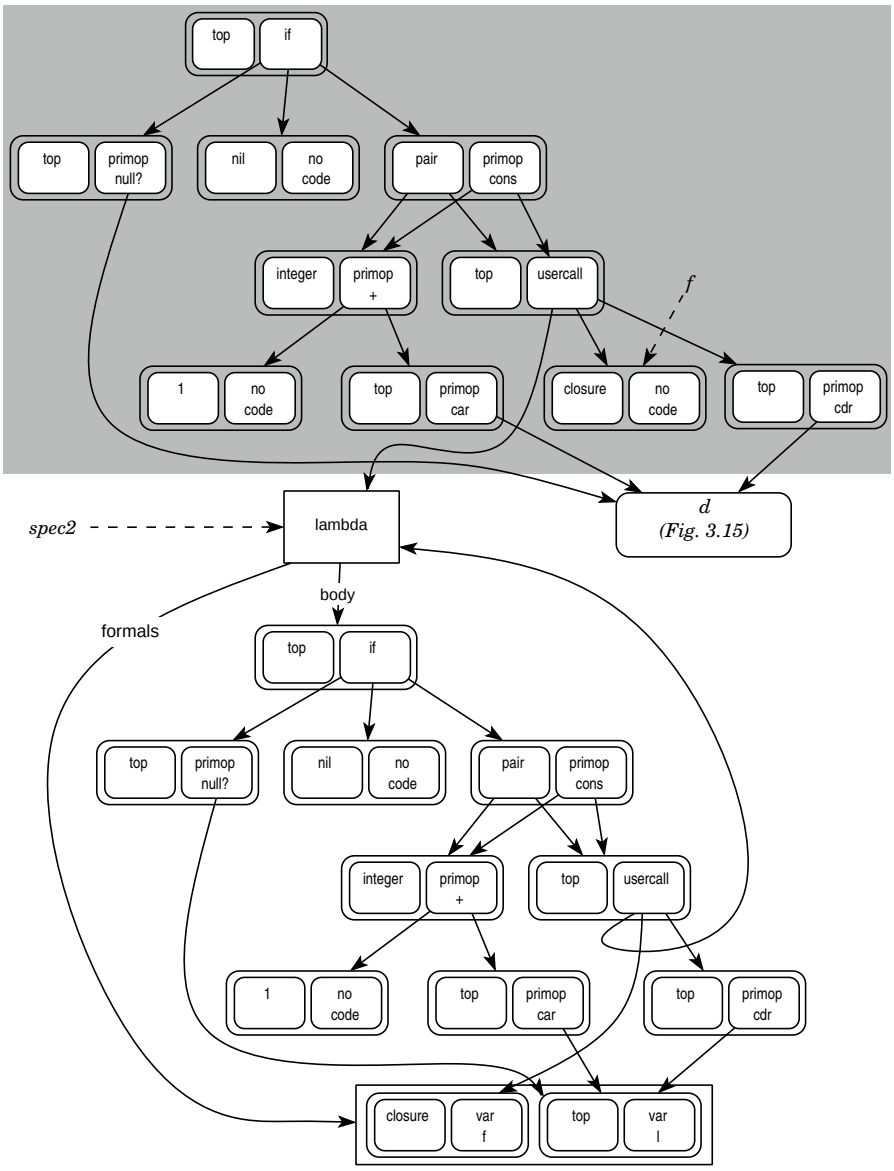


Figure 3.19: A preamble. All of the shaded graph nodes can be replaced by a single call, (*spec2 f d*).

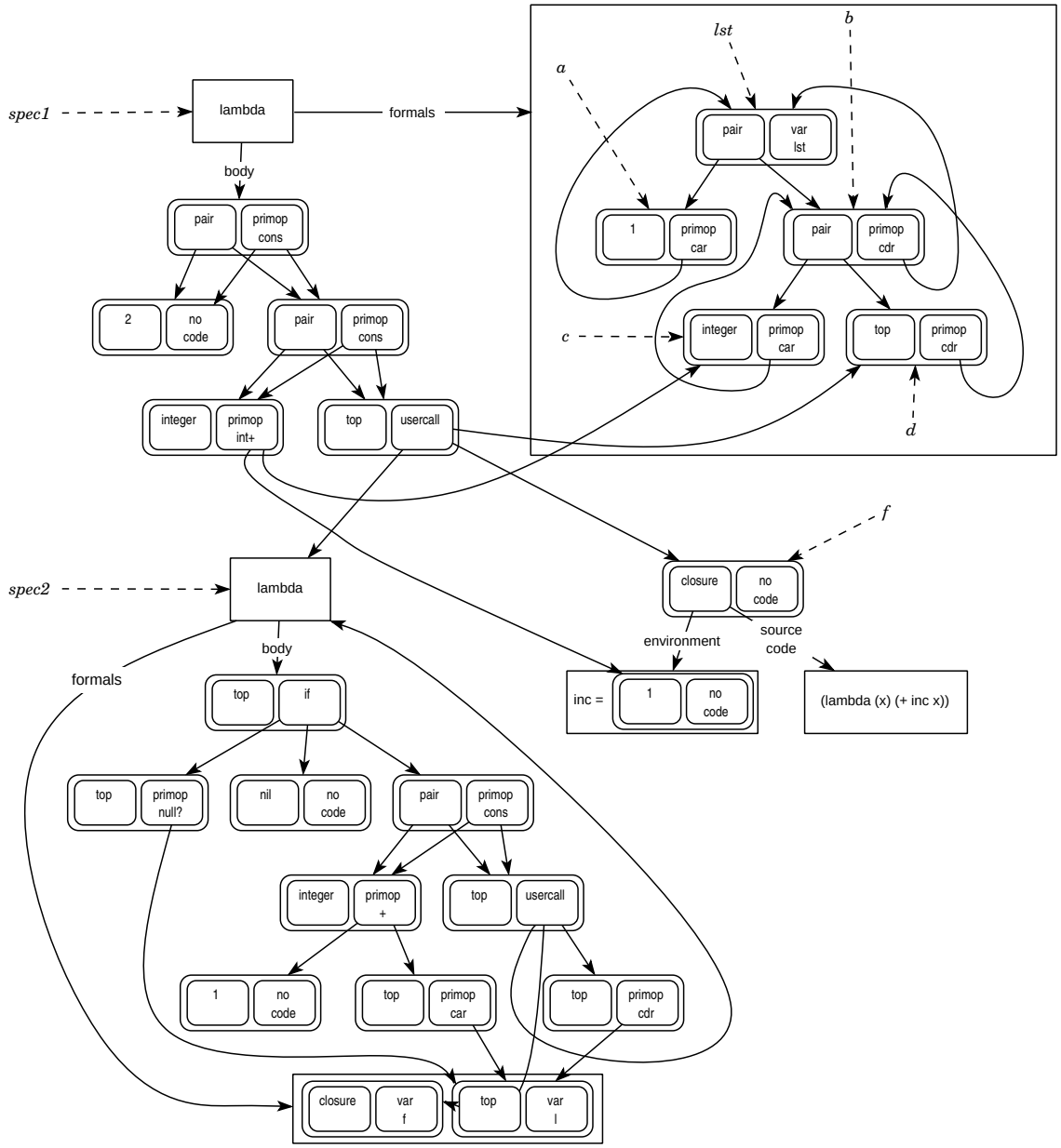


Figure 3.20: Final result of specializing `map1+` on `(1` $\top_{Integer}$ `.` $\top_{Val}$`)`

3.5 Summary

This chapter has described the input/output characteristics, internal data structures, and algorithms of the basic FUSE program specializer. The next three chapters describe extensions to this basic implementation for the purposes of computing better approximations (value specifications) of runtime values in both first-order and higher-order programs, and for improving the cache lookup strategy to avoid constructing redundant specializations. The Appendix gives more formal descriptions of both base FUSE and these extensions, as well as stating and proving some useful properties of these algorithms. We conclude this chapter with a brief discussion relating FUSE's implementation to that of prior online specializers.

3.5.1 Symbolic Values

Early online specializers such as REDFUN-2 [49] did not treat value specifications as first-class objects, which could be stored in data structures, environments, etc. Instead, REDFUN-2 used a data structure called a *Q-tuple* to record a set of type assertions at each program point. In some ways, Q-tuples were quite powerful (they could express disjoint unions and a weak form of negation), but they were not first-class objects, and could not be incorporated into closures or data structures. Schooler's *partials* [101] and Berlin's *placeholders* [8] are similar to our symbolic values, though Schooler's system uses a copying (rather than pointer-sharing) implementation of beta-substitution, causing duplication of code attributes. In both of these systems, the value specification and residual code expression attributes are sometimes compressed together; *e.g.,* the specializer deduces types of non-constant values from from the residual expressions rather than storing them in the symbolic value object. We believe Weise's FUSE [115] was the first system to recognize the need for this distinction (*c.f.* Section 3.2.1). Finally, Consel's "online parameterized partial evaluation" [29] framework represents values as tuples whose first element is either a constant value or a residual code expression, and whose remaining elements are typed attributes drawn from finite algebras. This increases the expressiveness of the type system, but still limits the flexibility of the code generator, which is denied the option of representing constant values as variable references or constructor invocations, rather than as quoted values.

3.5.2 Graphs

The use of graphical output representations in partial evaluation was pioneered by Berlin [8, 10], who took advantage of a graphical representation using *placeholder* objects to expose large amounts of parallelism. Perlin's work [85] uses what he terms *call graphs* as the target of a partial-evaluation-like transformation; rather than translating the graphs back to a textual form, his system interprets them in a memoizing manner, allowing programs to be incrementally re-evaluated as inputs change. Turchin's "supercompiler" [111, 112] uses a graphical representation for both source and residual programs, but his published work does not address code duplication issues. The graph structure and code generator described in this dissertation are descendants of those used in Weise's early FUSE systems [114, 115].

3.5.3 Termination

Early online program specializers relied exclusively on manual annotations for achieving termination. Some specializers [8, 85] avoid the problem entirely by promising termination only for programs where all control flow is determined only by known inputs, making it safe to unfold all procedure calls.[35] FUSE's stack-based recursion detection, first described in [115], is similar to those used in online specializers for other languages [112, 100]. The use of pairwise generalization rather than static argument abstraction for computing the arguments for specialization dates back to REDFUN-2 [49], which analyzed `GOTO` loops to determine which arguments varied and which remained constant. Pairwise generalization is now used by many online specializers [115, 112, 100, 107, 43]. More recent work on online termination methods includes finiteness analysis [45] (which, although designed for use in offline systems, also works in the online case), display-based recursion detection [69], and control-dependence-based approaches [70].

[35] This might seem naive, but is still useful. With the exception of an outer "convergence loop," most loops in numerical programs have known bounds, and can be completely unrolled by such methods. Even when the loop bounds are not known, one can still benefit by specializing the *bodies* of loops using such methods.

Chapter 4

Accuracy 1: Return Values

We have seen that it is desirable for a program specializer to perform as many reductions as possible at specialization time, relieving the residual program of the need to perform those reductions at runtime. We have also seen that, unlike an interpreter, which always has the information necessary to perform all reductions, a program specializer can only perform reductions when it has sufficient information to do so. The expression `(+ x 1.33)` cannot be reduced if nothing is known about the value of `x`, while if the specializer knows that `x=1.1`, it can reduce the expression to `2.43`. Even partial information can be useful: in a language with runtime typing, knowing that `x` is a floating-point number might allow the specializer to produce `(flonum+ x 1.33)`, eliminating some runtime tag checks; similarly, knowing part of a structured value (*e.g.,* `y`'s `car` is `4`, etc.) can be of use. Thus, it is important for the specializer to have access to as much information as possible during the specialization process.

The amount of information available to the specializer depends on two features: the complexity of the specializer's type system, which limits the detail with which values can be described, and the specializer's ability to infer and maintain this information. This dissertation is not about the former; our results and methods are largely independent of the particular type system used by the specializer. We are also not concerned with how the specializer infers type information from residual applications of primitives, as we consider this part of the type system as well. Instead, this chapter and the following chapter identify and treat two specific cases in which existing specializers "throw away" information available to them even though the information can be represented in their type systems, and using the information poses no risk of nontermination. This chapter discusses the sources of

```
(define (lookup-in-store name store)
  (if (eq? name (caar store))
      (cdar store)
      (lookup-in-store name (cdr store))))

(define (update-store store name new-value)
  (let ((binding (car store)))
    (if (eq? (car binding) name)
        (cons (cons name new-value) (cdr store))
        (cons binding (update-store (cdr store) name new-value)))))

(define (process-updates updates store)
  (if (null? updates)
      store
      (process-updates
        (cdr updates)
        (update-store store (caar updates) (cdar updates) store))))
```

Figure 4.1: Code to access and update (functionally) a store

information loss (both necessary and unnecessary) during the specialization of first-order programs[1], and gives an algorithm to avoid the unnecessary losses, while Chapter 5 discusses and solves a corresponding information loss problem for higher-order programs.

As a simple example of where existing methods fall short, consider code that maintains a store represented as an association list (Figure 4.1). The procedures `lookup-in-store` and `update-store` access and update (functionally) a store, while the procedure `process-updates` accepts a list of update requests and a store, and returns the resulting store. Similar code might be present in interpreters, simulators or other programs that maintain a global state of this form.

Specializing `update-store` on a `store` of the form `((a . `$\top_{Val}$`) (b . `$\top_{Val}$`))`[2] and an unknown `name` and `value` will return a store of the same form. This information (the shape

[1] Actually, our analysis and solution will apply to all programs which, when specialized, result in first-order residual programs (*i.e.*, as long as all first-class functions are inlined, we don't have to treat them specially when computing approximations). Thus, we will limit ourselves to programs where all `ABSCALL` forms are unfoldable.

[2] A note on terminology: we use terms like "any value," "any integer," and 42 to describe specialization-time approximations of runtime values, and use $\top_{Val}$, $\top_{Integer}$, and `42` to describe representations of those approximations. Thus, we can say either "a pair whose `car` is any value and whose `cdr` is any integer," or $(\top_{Val} \cdot \top_{Integer})$.

of the store) is important, because it will allow subsequent calls to `lookup-in-store` to be reduced to either a decision tree (when `name` is unknown) or to a simple sequence of `car` and `cdr` operations to index into the store (when `name` is known). Specializing `process-updates` on the same store and a unknown `updates` list should also return a store of the same form; if one update preserves the "shape" of the store, so should an arbitrary number of updates.

To the best of our knowledge, all existing (published) specialization techniques fail to compute accurate approximations to the values returned by (residual calls to) specializations, *even when their type systems are capable of representing such values.* Instead of computing an approximation, such specializers always use the approximation "any value," which is always safe, but often not very accurate. In the case of specializing `process-updates`, using "any value" instead of `((a .` $\top_{Val}$`) (b .` $\top_{Val}$`))` as the return value will cause subsequent invocations of `lookup-in-store` to be specialized as residual loops, even though the lookups could have been resolved at specialization time. The method described in this chapter, which we call *Return Value Analysis*, overcomes this problem, and computes the desired return approximation for `process-updates`.

This chapter consists of six sections. Section 4.1 describes several ways in which information is lost during the specialization process, and categorizes them into necessary and unnecessary losses; Section 4.2 introduces two programs whose specialization suffers due to unnecessary losses. In Sections 4.3 and 4.4, we describe an algorithm for overcoming these losses, and its implementation in FUSE. After this, we revisit the example programs and show that our implementation produces the desired specialized programs. We conclude with a discussion of related work and directions for future work.

4.1 Sources of Information Loss

Information is lost in many ways during the specialization process. Some are inherent in the nature of specialization itself: the need to represent an infinite number of runtime values with a finite specialization-time type system, and the need to perform generalization on pairs of specialization-time values. Others, however, such as the use of overly general approximations to return values, are consequences of implementation decisions are need not take place. In this section, we describe these sources of information loss in more detail.

4.1.1 Type System

The values manipulated by the specializer are approximations that represent runtime values; for instance, the return value computed for a residual code expression must approximate *all* values that could be returned by that expression at runtime. The degree to which these approximations are accurate depends on the type system; information is lost when the type system is insufficiently precise to denote a particular set of possible runtime values exactly. Such information loss may lead to unnecessary residualization. For example, consider the program fragment

```
(let ((z (cons x y)))
   ...
   (if (integer? (car z)) ... ...))
```

where `x` and `y` are known to be integers. When the `cons` expression's return value is approximated by "a pair whose `car` and `cdr` are integers," (in FUSE terms, the value specification $(\top_{Integer} \cdot \top_{Integer})$) the `if` expression will be seen to be reducible, and will be reduced. Otherwise, if the return value is approximated by "a pair whose `car` is any value and whose `cdr` is any value," $((\top_{Val} \cdot \top_{Val}))$ or by "any value," $(\top_{Val})$ the `if` will not be seen to be reducible, and will not be reduced.

Clearly, a type system supporting finer distinctions will be capable of representing more information about runtime values, allowing for more accurate specialization.

4.1.2 Generalization

Whenever the specializer is forced to generalize (compute an upper bound of) two approximations, it can lose information. Part of this loss is fundamental: the least upper bound of `3` and `4` is "`3` or `4`;" there is less information because the specializer no longer knows which value will appear at runtime. The other component of this loss is, once again, the imprecision of the type system, which may be unable to denote the least upper bound exactly (*i.e.,* for value specifications a and b, it is not generally the case that $(C\,(a \sqcup b)) = (C\,a) \cup (C\,b)$, but only that $(C\,(a \sqcup b)) \supseteq (C\,a) \cup (C\,b)$). If generalizing `3` and `4` yields $\top_{Integer}$ or $\top_{Val}$, an additional loss has occurred, since the specializer would no longer be able determine, for example, that the value in question is less than `5`.

Such generalization occurs in two places in most online specializers. The first is when computing the return value of a residual `if` statement, which must approximate the values

returned by both arms (for example, see the "online parameterized partial evaluation" semantics on p. 99 of [29], the `if`-handler code on p. 11 of [116], or the description in Section 3.3.1). The second is when computing the argument values to be used when building a specialization (discussions of generalization can be found in Section 3 of [115], [112], and Sections 2.3.2 and 3.3.4 of this dissertation.) In both cases, some loss of information is unavoidable, but can be minimized through the use of a more precise type system which can compute less overly general upper bounds.

4.1.3 Return Values

The losses described above are inherent in the nature of specialization: all type systems are imprecise, and all specializers must compute generalizations.[3]

However, information loss in existing specializers is not limited to these cases. *In particular, existing specialization techniques for first-order programs always use $\top_{Val}$ as the approximation to the value returned from a residual procedure call, even when their type system might be able to represent a better approximation.* For example, when the recursive `length` function[4]

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

is specialized on `lst=`$\top_{Val}$, its return value is always an integer. This could be used to specialize the application of `+`, replacing it with `int+`, possibly avoiding some runtime type tests. However, existing systems that can represent the type $\top_{Integer}$ still do not achieve this result, because the recursive call to `length` is viewed as returning $\top_{Val}$.

[3] There is no explicit generalization code in offline specializers because the BTA finds all places where generalization would be necessary, and forces those values to be treated as "dynamic," (*i.e.,* as though they evaluated to $\top_{Val}$ at specialization time) which gives the same effect as generalization, though in a less accurate manner.

[4] This example, like many of the examples which follow, assumes a specializer that computes type approximations for scalars. This is for purposes of exposition, since it allows us to build smaller examples. Our argument and techniques are equally valid for specializers that don't, because Scheme programmers often use structured data to build "ad hoc" representations of types. Often, the information which we wish to preserve about a return value is its structure, and the values of some subparts of that structure (*e.g.,* preserving the ordering and values of the names within the store in the interpreter example of Section 4.2.1).

```
Program  ::=  (program (pars Id*) (dec Id*) (procs Command*) Command)
Command  ::=  (:= Id Exp) |
              (if Exp Exp Exp) |
              (while Exp Command) |
              (begin Command*) |
              (call Id)
Exp      ::=  (quote Exp) |
              (car Exp) |
              (cdr Exp) |
              (not Exp) |
              (atom Exp) |
              (equal Exp Exp)
```

Figure 4.2: Syntax of the MP+ language

4.2 Examples

In this section, we give two more realistic examples of programs where existing program specializers fail to produce good-quality specializations because of information loss during the specialization process. These losses are *not* inherent in the nature of specialization, but are avoidable.

4.2.1 Interpreter Example

Our first example is drawn from the domain of interpreters, which are common targets for specialization. The "MP+" language, whose syntax is shown in Figure 4.2, is an extension of the by-now-canonical "MP" language, first used as an example by Sestoft [102]. Programs consist of declarations of input variables, local variables, and procedures, followed by a body, which is one of five commands (`:=`, `begin`, `if`, `while` and `call`). Commands may contain constants, as well as a variety of unary and binary expressions (`cons`, `car`, `cdr`, `not`, `atom`, and `equal`).

Figure 4.3 shows a fragment of a direct-style, recursive descent interpreter for this language. The interpreter begins by building an initial store, represented as an association list, mapping each name declared in the `pars` section to the corresponding element of the input, and each name declared in the `vars` section to the empty list. The values (`cdr`s) of bindings in this store are altered as interpretation proceeds, but the names (`car`s) never

```
(define mp
  (letrec
     ((init-store ...)
      (mp-command
       (lambda (com decls store)
         (let ((token (car com)) (rest (cdr com)))
           (cond
            ((eq? token ':=)
             (let ((new-value (mp-exp (cadr rest) store)))
             (update store (car rest) new-value)))
            ((eq? token 'if)
             (if (not (null? (mp-exp (car rest) store)))
                 (mp-command (cadr rest) decls store)
                 (mp-command (caddr rest) decls store)))
            ((eq? token 'call) (mp-call com decls store))
            ((eq? token 'while) (mp-while com decls store))
            (else ;(eq? token 'begin)
             (mp-begin rest decls store))))))
      (mp-call
       (lambda (com decls store)
         (let ((procname (cadr com)))
           (mp-command (lookup-proc procname decls)
                       decls
                       store))))
      (mp-begin
       (lambda (coms decls store)
         (if (null? coms)
             store
             (mp-begin (cdr coms) decls (mp-command (car coms) decls store)))))
      (mp-while
       (lambda (com decls store)
         (if (mp-exp (cadr com) store)
             (mp-while com decls (mp-command (caddr com) decls store))
             store)))
      (mp-exp ...)
      (lookup-proc ...)
      (update ...)
      (lookup ...)
      (main ...)
     main))
```

Figure 4.3: Fragment of an interpreter for MP+

change. If this interpreter is specialized with respect to a known program and unknown input, we would expect that the store will appear in the residual program, since it implements variable access. However, since the names in the store never change, all residual accesses to the store should be fully unfolded into sequences of `car` and `cdr` operations. There should be no residual loops that search for the correct binding in the store. A postpass could further improve the program by converting the list accesses to tuple accesses, or by arity raising the store to convert its elements into distinct variables. For instance, specializing the interpreter on the program

```
(program (pars x)
         (dec y)
         (procs)
         (begin
           (while x
             (begin
               (:= y (cons '1 (cons '1 y)))
               (:= x (cdr x))))
           (:= y (cons '1 y))))
```

(which computes $y = 2x + 1$ for numbers represented in unary notation) and an unknown input should yield a specialization like the one shown in Figure 4.4. Note that accesses and updates to the store have been replaced by open-coded strings of `car` and `cdr` operations: the names in the store are not used by the residual program at all, allowing various optimizations such as arity raising.

Unfortunately, this isn't what most specializers produce (*c.f.* Figure 4.5). The `while` loop in the program causes the specializer to build a residual version of the `mp-while` procedure which recursively invokes itself to implement the `while` loop. Existing specializers don't compute approximations to the value returned by a residual procedure call, but just use the most general approximation "any value," which is always valid, but not very precise, since partial information about the return value may be available at specialization time. In this example, that decision means that the return value approximation from both the original and recursive calls to `mp-while` is $\top_{Val}$ instead of `((x .` $\top_{Val}$`) (y .` $\top_{Val}$`))`, meaning that any accesses to the store after the program exits the while loop (*e.g.,* procedure `lookup6` and `update5` in Figure 4.5) will have to be residualized as loops that search the store, instead of as open-coded accessors. This makes arity raising impossible. In this particular example, only the access and update to `y` in `(:= y (cons '1 y))` are residualized in this manner, but in general, all store accesses after a loop exit will be residualized

```
(letrec
    ((mp-while35
      (lambda (store)
        (if (cdr (car store))
            (mp-while35
             (cons (cons 'x (cdr (cdr (car store))))
                   (cons (cons 'y (cons '1 (cons '1 (cdr (car (cdr store))))))
                         '()))
             store)))
     (main34
      (lambda (input)
        (let ((temp36
               (mp-while35
                (cons (cons 'x (car input)) '((y))))))
          (cons (car temp36)
                (cons (cons 'y (cons '1 (cdr (car (cdr temp36))))) '())))))))
  main34)
```

Figure 4.4: Desired result of specializing MP+ interpreter on program. Completely static formal and actual parameters have been eliminated, but arity raising has not been performed. The boxed code implements the statement `(:= y (cons '1 y))`; note that the store access and update have been open-coded.

sub-optimally.

The situation is often worse, for two reasons. First, not only is the structure of the store lost, but any type information about values in the store is lost as well. There isn't any such information in this example, but there might well be in others (such as interpreters for runtime-typed languages, which tag the values in the store with their types). Second, in this program, the structure of the store was retained in the body of the loop because the residual version of `mp-while` is tail-recursive (the recursive call to `mp-command` gets inlined, and is not a factor). That is, the body of `mp-while` doesn't use the return value from `mp-while`, so using a bad approximation to that return value doesn't affect the body. However, were the program to contain nested `while` loops, all code after the innermost loop would lose the structure of the store. Furthermore, as we shall see later (*c.f.* Section 5.5), specialization of the interpreter on MP+ programs which use procedures (*e.g.,* contain MP `call` statements) can cause the construction of truly recursive loops, whose return value *is* used by their bodies.

As we described in Section 2.4.2 (page 36), there are three common work-arounds for

```
(letrec
    ((mp-while7
      (lambda (store)
        (if
         (cdr (car store))
         (mp-while7
          (cons (cons 'x (cdr (cdr (car store))))
                (cons (cons 'y (cons '1 (cons '1 (cdr (car (cdr store)))))) '())))
          store)))
     (lookup6
      (lambda (store)
        (if (eq? (car (car store)) 'y)
            (cdr (car store))
            (lookup6 (cdr store)))))
     (update5
      (lambda (store val)
        (if
         (eq? (car (car store)) 'y)
         (cons (cons (car (car store)) val) (cdr store))
         (cons (car store) (update5 (cdr store) val)))))
     (main4
      (lambda (program input)
        (let ((temp8
               (mp-while7
                (cons (cons 'x (car input)) '((y))))))
          (update5 temp8 (cons '1 (lookup6 temp8))))))))
     main4)
```

Figure 4.5: Usual result of specializing MP+ interpreter on program. Completely static formal and actual parameters have been eliminated, but arity raising has not been performed. The boxed code implements the statement `(:= y (cons '1 y))`; note that the store access and update invoke specializations that search the store.

the problem illustrated by this example, all based transforming the program to avoid the need to pass the names upward:

1. Separate the store into a list of names and a list of values. The list of names is only passed downward, so that, when information is lost about upward-passed values, the list of names will not be lost. Now, when information is lost about upward-passed values, the list of names will not be lost. Of course, any values (or types or subparts of values) will be lost. This is a very common transformation; Mogensen [83] and Deniel et al [34] have succeeded in automating it under offline frameworks.

2. Rewrite the program into a purely tail-recursive form that only passes the store downward, never upward. This approach is taken by Mogensen [83] and by Launchbury [76], whose interpreters pass an extra argument containing the "rest" of the MP+ statements to be executed. Instead of returning a store when the loop is complete, `mp-while` exits by calling `mp-command` on the "next statement" and the store.

3. Perform the CPS [108] transformation on the program, as is done by Consel and Danvy [26]. This is a general transformation which is applicable to all programs; specializing the CPS converted form of the interpreter yields better specializations for many MP+ programs, including the addition program shown program above.

Each of these work-arounds has a cost: (1) only preserves the names in the store, not the values, (2) requires considerable hand-rewriting, and (3) introduces higher-order constructs, potentially complicating specialization and reducing polyvariance. Furthermore, for truly recursive programs, (3) merely trades one information loss problem for another; we will cover this in Chapter 5.

4.2.2 Integration Example

Our second example is drawn from another popular domain of programs: scientific and numerical computation. Consider an idealized routine for performing integration using a divide-and-conquer paradigm (Figure 4.6). On each iteration, the loop computes an approximation using the trapezoidal rule, then calls a predicate `good-enough?` that computes a more complex estimate and returns true when the two estimates are sufficiently close. Otherwise, the loop subdivides the interval, recursively computes solutions for the subintervals,

```
(letrec
    ((integrate-loop
      (lambda (fcn lhs rhs)
        (let ((guess (* (- rhs lhs) (/ (+ (fcn lhs) (fcn rhs)) 2))))
          (if (good-enough? fcn lhs rhs guess)
              guess
              (let ((mid (/ (+ lhs rhs) 2)))
                (+ (integrate-loop fcn lhs mid)
                   (integrate-loop fcn mid rhs))))))))
  integrate-loop)
```

Figure 4.6: Divide-and-conquer integration program

and sums them.[5]

We will specialize this loop with respect to a known function, but with unknown integration bounds (the types of the bounds may be known). We expect that the function and termination test will be unfolded, and that the various arithmetic operators in the function, predicate, and in the body of the integration loop, although left residual, will be specialized using the types of the integration bounds and the integration result. In particular, we expect that the `+` operator used to implement the sum of the subinterval estimates will be specialized on the same type as `fcn`'s result type. If we specialize `integrate-loop` on `fcn=(lambda (x) (* x x))` and `lhs` and `rhs` known to be numbers, we should get a specialization in which none of the arithmetic operators have been residualized in a form that type-check their operands (Figure 4.7). The sum of the subinterval estimates can be specialized in this manner because the recursive calls to `integrate-loop` are known to return numbers. This might allow a compiler to eliminate some tag checks.

Existing specializers, which cannot compute return types of residual calls, use the approximation "any value" for the values returned from the recursive calls, leading to a specialization in which operators depending on the types of the loop's formal parameters are still specialized, but the `+` operator which adds the subinterval estimates cannot be specialized (Figure 4.8).

At first, this might seem like a small price to pay; after all, the vast majority of the operators (including those in the predicate, whose expansion we have omitted for brevity)

[5]Of course, a real integration routine would do some extra parameter passing and, possibly, memoization to avoid redundant evaluations of `fcn`, but the point of the example here is to show a divide-and-conquer algorithm, not to teach numerical methods.

```
(letrec
    ((integrate-loop-10
      (lambda (lhs-8 rhs-7)
        (let ((guess-5
               (tc-* (tc-- rhs-7 lhs-8)
                     (tc-/ (tc-+ (tc-* lhs-8 lhs-8) (tc-* rhs-7 rhs-7)) 2))))
          (if
           <unfolded version of good-enough? omitted>
           guess-5
           (let ((mid-9 (tc-/ (tc-+ lhs-8 rhs-7) 2)))
             (tc-+
              (integrate-loop-10 lhs-8 mid-9)
              (integrate-loop-10 mid-9 rhs-7)))))))
  integrate-loop-10)
```

Figure 4.7: Desired result of specializing integration routine. We have omitted the unfolded residual version of the predicate `good-enough?`. Note that the addition of the subinterval estimates (recursive calls to `integrate-loop10`) is performed with a specialized addition operator, `tc-+` ("typed checked `+`").

```
(letrec
    ((integrate-loop-10
      (lambda (lhs-8 rhs-7)
        (let ((guess-5
               (tc-* (tc-- rhs-7 lhs-8)
                     (tc-/ (tc-+ (tc-* lhs-8 lhs-8) (tc-* rhs-7 rhs-7)) 2))))
          (if
           <unfolded version of good-enough? omitted>
           guess-5
           (let ((mid-9 (tc-/ (tc-+ lhs-8 rhs-7) 2)))
             (+
              (integrate-loop-10 lhs-8 mid-9)
              (integrate-loop-10 mid-9 rhs-7)))))))
  integrate-loop-10)
```

Figure 4.8: Usual result of specializing integration routine. We have omitted the unfolded residual version of the predicate `good-enough?`. Note that the addition of the subinterval estimates is performed with the general addition operator, `+`.

---

are properly specialized. Only one operator is improperly specialized, leaving only one unnecessary tag test. Although this is the case in this simple example, it is not so in general. Often, numeric programs use more complex data types than the built-in scalar types; for instance, one might use an object-oriented representation for complex numbers such as that in Section 2.3 of [1]. In such cases, which may involve multiple levels of dispatch, it is important to be able to resolve the method dispatch for operators like `+` at specialization time, since it might involve many operations at runtime. Furthermore, it may allow the fields used for tagging such ad hoc types to be removed by transformations like arity raising, which is not possible if their contents (the tag symbols) are still used in comparison operations in the residual program.

Also, one might be led to believe that this problem could be solved by simple scalar type inference in a postpass or in the compiler. Such an inferencer would deduce that `integrate-loop` returns a number, and could thus replace the general `+` for adding the subinterval estimates with one optimized for numeric arguments. There are two problems with this approach. First, if some other expression (such as a call to `number?`) depends on the return value, it will not be reduced by the postpass, which is not a specializer, and cannot perform arbitrary reductions, build specializations, etc. Second, a scalar type inferencer would not be able to optimize the program when ad hoc types, such as those described in the previous paragraph, are used. It might be able to deduce that the return value is a pair whose `car` is a symbol and whose `cdr` is a number, but it would not be able to deduce that the head is the symbol `'polar-complex` and resolve the dispatch accordingly.

4.2.3 Commentary

In both of these examples, achieving a good specialization required computing accurate approximations to values returned by residual calls to specializations. Thus, both examples serve to motivate the mechanism that will be described in Section 4.3; we will revisit these examples and demonstrate that mechanism in action in Section 4.5.

Existing approaches to dealing with the problem of computing approximations of return values of residual procedure calls have concentrated on *avoiding* the problem rather than on solving it. That is, they rewrite the program in such a way that, even if the specializer computes overly general approximations to return values, it won't affect the quality of the specialization. The three work-arounds described in Section 4.2.1 are of this form; they either convert the program so that all information is passed downward, or so that any

upward-passed information can be lost without affecting the specialization. Of the three approaches, only the CPS transformation [108, 26] was fully general and automatable, but since it builds higher-order programs, it cannot be considered a solution for first-order specializers. Furthermore, for these examples, this transformation only serves to trade one information loss problem (approximations to return values) for another (approximations to parameter values). We will revisit these examples in CPS-converted form in Chapter 5.

We should also note that we have not presented benchmarks for the "desired" and "actual" specialized programs presented in this section. This is deliberate—for two reasons. First, the performance penalty due to information loss is heavily dependent on the inputs to the specialized program; *e.g.,* by increasing the number of statements after the `while` loop in the MP program, we can make the "actual" program perform arbitrarily poorly. Although we could choose some minimal slowdown (such as our example program, which performs only one access and one update to the store after its structure is lost), it's difficult to decide on a representative input. Second, and more importantly, most users of specializers would never tolerate the inefficient "actual" programs presented here; *e.g.,* no one would consider a specialization of an interpreter in which the structure of the store is lost to be acceptable. In practice, such programs are always rewritten (using the methods described above) so that even "lossy" specialization techniques will produce the desired results. Our primary goal in this research is to eliminate the need for this rewriting, not to achieve particular speedup figures.

4.3 Fixpoint Iteration Solution

Instead of rewriting the source program to avoid the consequences of overly general approximations to return values, we can instead choose to compute better approximations. This allows us to treat direct style programs without pre-transforming them, producing good results without human intervention or the introduction of higher-order constructs. This section describes the basics of *return value analysis*, an extension to the basic online specialization algorithm for this purpose. Section 4.4 describes how this

The basic idea is to associate a single value specification (element of *Val* with each specialization, denoting a conservative approximation to the set of values it could potentially return at runtime. We can do this by extending the specialization cache to return both a return value approximation and a specialized body, rather than just a specialized body. The

remainder of this section describes how the specializer computes and makes use of these approximations, and gives a small example.

4.3.1 Computing Return Value Approximations

Making use of return value approximations is simple; whenever a residual call to a specialization is constructed, we use the return value approximation associated with the specialization as the return value of the residual call, rather than using $\top_{Val}$, as we did in base FUSE.

Computing approximations for complete specializations is also straightforward. The base specializer already computes a value specification for every expression it constructs; the return value approximation is simply the value specification of the residual expression that is the body of the specialization!

The remaining problem is: what is the return value approximation of an incomplete specialization (*e.g.,* one that doesn't *have* a body yet, because it is still being constructed)? We could use $\top_{Val}$, but that would mean that any reductions inside the body depending on the return value of a recursive call would not be performable (since the recursive call is assumed to be capable of returning any value), and that the value specification of the body (used as the return value approximation once the specialization is compete) would be overly general.

Instead, we will start out with an overly specific approximation, and refine it into a sufficiently general one. That is, we view the specialization of the body as a monotonic function from an initial return value approximation to a new return value approximation,[6] and attempt to find the least fixed point of this function in the domain of value specifications, *Val*.

This can be accomplished by adding a unique bottom element to the specializer's type lattice (we'll call it $\bot_{Val}$) and using it as the initial approximation to the return value of a residual procedure call. This initial approximation is used in computing an approximation to the return value of the body of the procedure. If this new approximation differs from the previous one, the body is evaluated using this approximation to the residual call's value, again and again, until the approximation does not change. This is similar to the fixpointing solutions used in iterative dataflow analyses [2] and abstract interpretation frameworks

[6] As we shall see, since multiple specializations may be "in progress" at any given time, the construction of specialization's body is actually a function of the return value approximations of all incomplete specializations.

such as Binding Time Analysis [64, 83]. Our solution bears an even greater similarity to the Minimal Function Graph (MFG) framework of [61]; if we view the specializer's cache as mapping function names and parameter approximations to specializations and return value specifications (instead of just specializations, which is the usual case), then specialization is can viewed as finding a fixed point over the cache interpreted as a MFG.

4.3.2 Correctness and Termination

The basic intuition behind iterative fixpoint-finding algorithms is as follows. The initial return value approximation (call it v_0) is incorrect, so any recursive calls to the specialization constructed during the first unfolding of the body (returning the initial approximation) will be incorrect—thus, the new approximation v_1 after the first iteration will be correct for all executions of the specialization making no recursive calls. For each subsequent iteration, the new approximation v_{i+1} is correct for all executions making at most i nested recursive calls to the specialization. Once a fixed point (*e.g.,* a v_k such that $v_k = v_{k+1}$ is found, that approximation is good for any execution of the specialization.

The iteration process converges to a correct least fixed point v_k provided that (1) the approximation computed on each iteration is correct (w.r.t the assumptions implicit in the environment and specialization cache in which the body is specialized), (2) the approximation computed on each iteration rises monotonically in the type lattice, and (3) the type lattice contains no infinite ascending chains. Properties (1) and (3) are true for base FUSE, except that we must modify primitives to handle the value $\perp_{Val}$; property (2) will require some extra mechanism (*c.f.* Sections 4.4 and A.4).

4.3.3 Example

Assuming, for the moment, that the above restrictions are met, let us consider the specialization of the recursive `length` program:

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

Expression	Iter. 1	Iter. 2	Iter. 3
`x`	$\top_{Val}$	$\top_{Val}$	$\top_{Val}$
`(null? lst)`	$\top_{Val}$	$\top_{Val}$	$\top_{Val}$
`(cdr lst)`	$\top_{Val}$	$\top_{Val}$	$\top_{Val}$
`(length (cdr lst))`	$\bot_{Val}$	`0`	$\top_{Integer}$
`(+ 1 (length (cdr lst)))`	$\bot_{Val}$	`1`	$\top_{Integer}$
`(if (null? lst) 0 (+ 1 (length (cdr lst))))`	`0`	$\top_{Integer}$	$\top_{Integer}$
`(length y)`	`0`	$\top_{Integer}$	$\top_{Integer}$

Figure 4.9: Fixpoint iteration on `length` example

We would like to compute an approximation to the return value of the expression `(length y)` where `y`=$\top_{Val}$. Because the approximation of the `if` depends on the approximations of its arms' values, and the approximation of the `+`'s value depends on the approximation of `(length (cdr lst))`'s return value, we can see that computing the return value of `length y` requires computing the return value of `(length (cdr lst))`. In the residual program, however, both the initial and recursive calls to `length` will invoke the same specialization (because both calls bind `lst` to the same type, namely $\top_{Val}$). Thus, the return value of the specialization depends on itself. We have already seen that starting out with a safe initial approximation of $\top_{Val}$ yields unsatisfactory results; in the above, assuming that the recursive call (length (cdr lst)) returns $\top_{Val}$ will produce an approximation of $\top_{Val}$ for the entire body of the specialization. Instead, we will perform fixpoint iteration.

In Figure 4.9, rows represent source expressions, while columns represent iterations of the algorithm. The finite height assumption is maintained here by forcing the generalization of 0 and 1 to be "any integer;" if we were to allow disjoint unions such as "0 or 1," the analysis would not terminate.

On the first iteration, we assume that the specialization returns $\bot_{Val}$, and compute a return value of 0 under this assumption. Since the computed value isn't the same as the assumed one, we haven't found the fixed point yet, so we update the cache (actually, the value specification of the body of the *code-fcn* for this specialization in the cache) to assume a return value of 0. Given this assumption, the specializer computes a return value of $\top_{Integer}$. A subsequent cache update and recomputation of the return value also returns $\top_{Integer}$, so we're done. Any residual calls to this specialization will return a value

specification of $\top_{Integer}$, which correctly approximates the set of values returnable by the specialization at runtime.

4.4 Implementation of Return Value Analysis

This section informally describes the implementation of the fixed point algorithm for computing return values in FUSE. A more detailed description with more formal correctness and termination arguments is given in Section A.4.

4.4.1 Basics

Fixpoint iteration is implemented in FUSE by adding a bottom element to the type hierarchy and modifying the procedure call code in the specializer. The specializer, which formerly maintained an association between specializations and their residual code, must now maintain an association between specializations, their residual code, and the types of their return values.

Specialization proceeds as follows. When the specializer decides to build a specialization, it adds an entry associating the source procedure, the argument vector, and an initial type approximation of $\bot_{Val}$ to the cache. It then adds the bindings between formals and actuals to the environment, and recursively calls itself on the procedure's body. Subsequent attempts to specialize the same procedure on the same argument vector will return the type approximation from the cache. When the process of specializing the body is complete, the type approximation of the body's symbolic value will be compared with the cached approximation. If they are the same, the specialization is complete; otherwise, the cached approximation is updated, and the body is specialized again. This process proceeds until the new and old approximations are equal.

4.4.2 Technicalities

The simplified description given above omits several important technical points.

Dependencies

Fixpoint iteration necessitates a change to the control structure of FUSE as described in Chapter 3 and in [115]. Traditional abstract interpretation methods do not specify a particular control strategy; they simply keep recomputing the approximations of all program

points until they all converge. FUSE, however, already has a depth-first, interpreter-like control strategy, which we must modify somewhat. Before fixed point iteration, each specialization could be considered independently; each depended only on its arguments. Once a specialization was built and cached, it never had to be touched again. This is no longer the case, because one of a specialization's properties, its return value (and thus its residual code), may depend on the return value of another specialization. Consider the program

```
(define (a x)
  (if (pred1 x)
      x
      (if (pred2 x)
          (... (a (- x 1)) ...)
          (... (b x) ...))))

(define (b y)
  (if (pred3 y)
      (... (b (/ y 2)) ...)
      (... (a y) ...)))
```

where `a` is specialized on $\top_{Val}$. During the first iteration of the fixed point computation for this specialization, `b` will also be specialized on $\top_{Val}$. After the first iteration of `a`'s fixpoint loop, the return value approximation for `a`'s specialization may have changed, but the completed specialization of `b` may contain residual code that depends on the old value of `a`'s return value. Thus, even though the parameters to the specialization of `b` have not changed, the specialization must be rebuilt. FUSE accomplishes this by keeping track of dependencies between specializations.[7] In this case the specialization of `b` depends on the return value of the specialization of `a`, and vice versa. If either specialization's return value changes, the other specialization's body (and thus its return value) will be recomputed. Note that such recomputations need not begin again with $\bot_{Val}$ as the approximation to the specialization's return value, but may proceed from the previous approximation.

Monotonicity

The fixpointing algorithm requires that the return value approximation computed by unfolding the body of function be a monotonic function of the return values of all the specializations in the cache under which the unfolding is performed; this assures convergence

[7] The description in Section A.4 uses a simpler, less efficient, strategy, invalidating *all* specializations computed during an iteration i before starting iteration $i+1$, regardless of whether they referenced the incorrect return value in effect during i.

by forcing the approximation to rise on each iteration. This property is easy to assure for the evaluation of forms other than `USERCALL`; we state without proof that it is true for primitive operators, special forms other than `USERCALL`, and the specializer's generalization operation.

However, assuring monotonicity for `USERCALL` forms requires a modification to the specializer. The risk of nonmonotonicity arises from the fact that multiple specializations can be "in progress" at once. Consider two successive iterations of the fixpointing algorithm for some particular specialization; in particular, consider a `USERCALL` form that is specialized on both iterations. If, the `USERCALL` is invoked on more general arguments[8] on the latter iteration, then it may match a *different* specialization in the cache. If that specialization is currently under construction, it may have a (incorrect) return value approximation that is *more specific* than that of the specialization matched on the prior iteration. This violates monotonicity because the latter `USERCALL` must return a *more general* value.

We solve this by noting a property of programs, namely that if executing a function f on some number of concrete argument vectors (*e.g.,* real Scheme values) yields some number of different return values, it must be the case that executing f on some larger set of argument vectors must yield all of the same return values, and possibly more. From this, we can conclude that the return value approximation of a specialization of f on argument specification a should be more general that the return value specifications of all specializations of f on a' where $a' \sqsubseteq a$. We achieve monotonicity by modifying the cache lookup function $(c\ f\ a)$ to return the least upper bound of the return value specifications of all specializations of f on arguments $a' \sqsubseteq a$. Since the implementation maintains a separate cache for each source function, and the number of specializations of each function is small, the cost of walking the cache to compute the least upper bound is quite low.

A similar problem arises if a `USERCALL` is unfolded on one iteration, and specialized on the next. If the specialization invoked is incomplete, it may have a return value which is more specific than that returned by the unfolding, which would violate monotonicity. One solution (used by the implementation in Section A.4) is to treat all unfoldings as though they were specializations, creating $\langle argument, return\ value \rangle$ associations in the cache. This allows our modified lookup function to enforce monotonicity, but at the cost of significantly slowing cache lookup, because some source functions may be unfolded many times. We can limit this cost via the following heuristic: we need only cache the return values of unfoldings

[8] This must be the case, if everything else is monotonic.

on argument values that are more specific than the argument values of some "in-progress" specialization of the same function. This requires maintaining an explicit representation of the set of specializations that are "in progress;" in the real implementation, this is easily derived from the termination mechanism's abstract call stack.

It should be noted that both of these cases are extremely rare in practice; except for custom test cases, we have yet to encounter a source program for which either of the above mechanisms was necessary. Situations of this sort require that a function invoke itself recursively on arguments computed from the results of another recursive call, such that the "inner" recursive call passes more specific arguments than the original call. Interpreters often exhibit this recursion pattern (*e.g.,* `eval` of `let` calls itself recursively to evaluate an initializer, then calls itself recursively on the body and an environment derived from the result of evaluating the initializer), but, typically, the arguments on each of the recursive calls are incomparable (since the expression argument is a different known value each time). Other situations, such as specializing `map` on a function `f` that invokes `map` on `f` (*e.g.,* tree traversal), are typically not problematic because either (1) the arguments to `map` are known, and both calls are unfoldable, or (2) the arguments to `map` are equally unknown at both call sites, and the same specialization is invoked at both cases.

Termination

The termination of the fixpoint iteration process (for computing the return value approximation for a particular specialization) depends on two factors: the monotonicity of the computation of return value approximations, and the finite height of the type lattice. If both of these constraints are met, then the fixpoint will be found in a finite number of iterations. Thus, the iteration process will terminate, *provided that each individual iteration terminates.*

The termination of an individual iteration (which builds a specialization using the return value approximation found by the previous iteration) is dependent on the specializer's mechanisms for avoiding infinite unfolding and the building of infinite specializations; if either of these occurs, control will never return to the fixpoint loop. This may occur if the termination method in use allows nontermination on programs where entry to a "statically

controlled" infinite loop is guarded by a conditional that can't be decided at specialization time.[9]

Since monotonicity was assured above, we need only address the phenomenon of *infinitely ascending chains*, in which each successive approximation is higher in the lattice, *ad infinitum*. For instance, fixpoint iteration on the function

```
(define (ones)
  (cons 1 (ones)))
```

might return the infinite sequence of approximations

$\perp_{Val}$, `(1 . `$\perp_{Val}$`)`, `(1 . (1 . `$\perp_{Val}$`))`, ...

FUSE's type system contains no disjoint unions, so the only types that could lead to an infinite chain of approximations are pairs (because they can contain other pairs), and closures (because their environments can contain pairs and closures). We address closures by forbidding return value approximations containing closures (we just raise such approximations to $\top_{Val}$)—this implies no loss of accuracy, since without first-class environments, the specializer has no way of instantiating a closure returned by a residual function call (*c.f.* page 64).

Infinite ascending chains of pairs can be avoided by construction. That is, all we have to do is prevent the bottom element $\perp_{Val}$ from appearing in any pair. That is, `cons` applied to `1` and $\perp_{Val}$ returns $\perp_{Val}$ instead of `(1 . `$\perp_{Val}$`)`.[10] This guarantees that, after the first iteration, the return value approximation is either $\perp_{Val}$, in which case the specializer has proven that the specialization doesn't terminate, and can return $\perp_{Val}$ with a clear conscience, or it contains no occurrences of $\perp_{Val}$ at all. In this latter case, termination is assured. First, the approximation will contain a finite number of pairs because the first iteration terminated. That is, since the specialization of the body terminated, it only executed the `cons` primitive a finite number of times. The only pairs that may appear in the body's approximation are

[9] Problems such as dynamically controlled loops with increasing static parameters are faced by all specializers. It is important to note that the difficulty faced in building a finite specialization of a procedure is no easier or more difficult with our fixpoint mechanism in place than without it. With respect to ensuring termination, the fixpoint mechanism's only responsibility is to iterate a finite number of times; the finiteness of the specialization(s) constructed on each iteration is the responsibility of the specializer's termination mechanism, whatever that may be.

[10] Another way to say this is that the domain *Val* of value specifications is constructed using *strict products*.

those from executing `cons` (which happened a finite number of times), those in the body's free variables (which are finite because only a finite number of specializations are built), and those in the return approximations of any specializations invoked by the body (the number of such specializations is finite, as is the size of their return approximations). Second, if the first iteration returns an approximation containing a finite number of pairs, the height of the lattice above that approximation is finite, because all values above *any* pair in the lattice contain an equal or smaller number of pairs (this would not be the case if we were allowed to put the bottom element in pairs; *e.g.,* `(1 . (1 .` $\perp_{Val}$`))` dominates `(1 .` $\perp_{Val}$`)`). This method works because Scheme is a strict language; if it were lazy, we would be unable to restrict the lattice in this manner.

Even if this construction is not used (for instance, the system described in [92] allows the bottom element to appear in pairs), infinite ascending chains are rarely a problem because most useful residual programs contain only well-founded loops. Well-founded loops always have base cases, which will necessarily contain a finite number of pairs, and no bottom elements. When such a base case is generalized with the recursive cases(s) of the loop, it establishes a finite length, bottom-free lower bound on the final approximation—once this is achieved, termination follows because such an approximation is only a finite distance below the top element of the lattice. Although the specializer is perfectly willing to construct residual loops that are not well founded (for instance, if an interpreter is specialized on a program containing an infinite loop, the specializer will build an infinite residual loop to implement it), such loops are rare in practice.

4.4.3 Cost

We have found that, in most cases, the fixed point is computed within two or three iterations; this is most likely due to the simplicity of the scalar type hierarchy. For programs without infinite loops, the fixpoint computation must perform at least two iterations (*i.e.,* the first iteration returns the approximation, and the second returns the same approximation). This, unfortunately, imposes a minimum factor-of-two performance penalty—in fixed point computations, the last iteration never accomplishes any work, but merely serves as a termination test. For procedures which are only ever called in tail position (*i.e.,* their return values are never used by any computation), such as those in Mogensen's downward-passing MP interpreter [83], there is no need to compute a return approximation at all, since it

won't ever be used. Unfortunately, at the point when a specializer is building a specialization, it cannot know how its return value will be used. This extra expense might be avoided through the use of lazy specialization techniques, but these come with their own overhead, and have not been been investigated in much detail (Launchbury [77] describes the use of lazy techniques to reduce representational overhead, but that's a different problem). In Section 5.4, we will describe another algorithm that does not impose this "extra iteration" penalty.

Because of mutual recursion, the cost of fixpoint iteration is potentially multiplicative in the depth of mutually recursive call paths in the residual program. For instance, in the program we saw above, recomputing one specialization may lead to the recomputation of another. In practice, this penalty is small, because (1) the number of mutually recursive procedures in *residual* programs is quite small,[11] and (2) because the recomputations, starting higher in the lattice, often converge faster than the original computations did.

4.5 Examples Revisited

In this section, we demonstrate our fixpoint iteration mechanism on the examples of Section 4.2.

4.5.1 Interpreter Example

Consider specializing the interpreter of Figure 4.3 on the program

```
(program (pars x)
         (dec y)
         (procs)
         (begin
           (while x
             (begin
               (:= y (cons '1 (cons '1 y)))
               (:= x (cdr x))))
           (:= y (cons '1 y))))
```

[11] Under FUSE's termination criterion, a mutually recursive *source* procedure must call both itself and one of its callers in order to be residualized. In many mutual recursions, one of the members merely "calls back" to its caller, and induces no loop of its own. Consider a Scheme interpreter, in which `eval` and `apply` are mutually recursive, but only `eval` is self-recursive. No specializations of `apply` will be constructed; instead, it will be unfolded in the body of the specialization of `eval`, yielding a residual program without mutual recursion.

and an unknown input. The specializer will build a specialization of the function `mp-while` on the expression

```
(while x
  (begin
    (:= y (cons '1 (cons '1 y)))
    (:= x (cdr x))))
```

an empty list of declarations, and a store of the form `((x . `$\top_{Val}$`) (y . `$\top_{Val}$`))`, resulting in a specialization like

```
(lambda (store)
  (if (cdr (car store))
      (mp-while35
        (cons (cons 'x (cdr (cdr (car store))))
              (cons (cons 'y (cons '1 (cons '1 (cdr (car (cdr store)))))) '()))
       store))
```

The initial approximation to the return value of the recursive call is $\bot_{Val}$. Thus, the approximation for the `if`, and thus the procedure body, is the generalization (least upper bound) of the approximation to the store and $\bot_{Val}$, which is `((x . `$\top_{Val}$`) (y . `$\top_{Val}$`))`. A second iteration of the fixpoint algorithm uses this as the approximation to the return value of the recursive call, which, after generalization with the approximation to the store, is the same as the previous approximation. Thus, when `mp-command` is unfolded on the expression `(:= y (cons '1 y))`, the approximation to the store is `((x . `$\top_{Val}$`) (y .`$\top_{Val}$`))`, allowing the calls to `lookup` and `update` to be unfolded. The final code is that shown in Figure 4.4.

4.5.2 Integration Example

Consider specializing the program of Figure 4.6 on `fcn=(lambda (x) (* x x))`, `lhs=`$\top_{Number}$, `rhs=`$\top_{Number}$. Thus, the approximation to `guess` will be "any number," and all of the arithmetic operators in the expressions for computing `guess` and `mid` will be residualized in a form that takes advantage of their arguments' being numbers. The recursive calls to `integrate-loop` are initially approximated by the bottom element, $\bot_{Val}$; their sum gets the same approximation. The approximation for the `if` is the generalization of $\top_{Num}$ and $\bot_{Val}$, which is $\top_{Num}$; this becomes the approximation the return value of the specialization of `integrate-loop`. On the second iteration, this approximation is used for the return values of both recursive calls, allowing the `+` operator which adds these values

```
(letrec
    ((integrate-loop
      (lambda (lhs rhs)
        (let ((guess (* (- rhs lhs) (/ (+ (fcn lhs) (fcn rhs)) 2))))
          (if (good-enough? lhs rhs guess)
              guess
              (let ((mid (/ (+ lhs rhs) 2)))
                (+ (integrate-loop lhs mid)
                   (integrate-loop mid rhs))))))))
     (fcn (lambda (x) (* x x))))
  integrate-loop)
```

Figure 4.10: First-order divide-and-conquer integration program

to be specialized on numbers, and return $\top_{Num}$. The `if` expression, and thus the body of the specialization, now returns $\top_{Num}$; since the approximation is unchanged, the fixpoint iteration terminates. The resulting residual program is identical to that shown in Figure 4.7.

4.6 Related Work

This section describes related work in program specialization, binding time improvement, type inference, and specialization-based compiler technology.

4.6.1 Specializers

Although no existing specializer performs fixpoint iteration to compute return value approximations, a variety of techniques have been used to improve the quality of specialization.

Haraldsson's REDFUN-2 [49], computes and propagates type information during specialization. For each residual expression, the specializer computes either a set of concrete values that the expression can return at runtime, a set of concrete values that it cannot return at runtime, or a scalar type descriptor, such as `SEXPR`, `INTEGER`, or `STRING`. REDFUN-2 goes further than FUSE in maintaining sets of concrete values, but its approximations, or *q-tuples* are weaker than symbolic values in that they denote properties of expressions, rather than values, and cannot be included in structured data or closures. REDFUN-2 only automatically derives type information from residual function calls in very restricted situations.

The online systems of Berlin [8] and Schooler [101] propagate information downward using *placeholders* and *partials*, respectively, both of which are similar to FUSE's symbolic values. However, these systems fail to propagate automatically derived type information upward out of `if` expressions or residual function calls.

The parameterized partial evaluation framework of Consel and Khoo [29] is a user-extensible type system for program specialization which can infer and maintain "static information" drawn from finite semantic algebras. Its online variant performs generalization to compute return values for `if` expressions, but its behavior with respect to return values of residual calls and parameters to specializations of first-class procedures is unspecified. Its offline variant cannot perform such generalization because of the need to make all reduce/residualize decisions in advance.

In the area of offline specialization, the partially static binding time analyses of Mogensen [83] and Consel [22], the projection-based analysis of Launchbury [76], and the higher-order binding time analyses of Bondorf [11], Mogensen [83], and Consel [23] reason about structured and function types, allowing offline specializers to make use of more information than the simple scalar BTA of MIX [64]. However, it should be noted that these analyses only produce descriptions of specialization-time structures, not of runtime structures. Online specializers like FUSE don't need to build recursive descriptions of such values, but instead simply operate on them. Similarly, binding time analysis can propagate information out of conditionals only when the test is static, whereas FUSE can do this in both the static and dynamic cases. Propagation of information out of residual procedure calls and into residual first-class procedures has not been implemented; based on the analysis of Chapter 2, we believe it will be difficult to perform these optimizations in an offline framework.

4.6.2 Binding Time Improvement

The class of program transformations applied to programs in order to improve the quality of residual programs obtained when they are specialized is called "binding time improvements." These transformations have been investigated mainly in the context of offline specialization, but can be useful under online frameworks as well.

Binding time improvement via the replication of code is common practice in writing interpreters to be specialized via offline means (for some example interpreters written in this style, see [12]). Automating such replication was first suggested by Mogensen [83]. Using the

CPS transformation to perform the same task was first suggested by Consel and Danvy [26] and has been used to improve the quality of specialization in several examples [24, 71]. other manual transformations, such as eta-conversion, are also common; Holst and Hughes [55] suggest a possible means of automating such transformations.

4.6.3 Type Inference

The techniques we use for computing approximations to return values and specialization parameters are similar to those used in abstract interpretation-based type inference.

The fixpoint iteration solution described in Section 4.3 is similar to the Minimal Function Graph analysis presented in [61], which computes sets of (input approximation, output approximation) pairs for each function in a first-order program. Although MFG techniques have been used in Binding Time Analysis [83], we believe that ours is the first application to the specialization phase.

Young and O'Keefe's *type evaluator* [118] is very similar to FUSE, cannot be considered to be a program specializer because it doesn't build specializations. The type evaluator discovers types (including recursive types) using a variety of techniques, including fixpointing and generalization as used in FUSE. Unlike FUSE, however, the type analysis performed by the type evaluator is monovariant in the sense that a polymorphic formal parameter of a function will be assigned the least upper bound of the types of the corresponding actuals from all calls to the function, while a polyvariant type analysis would be free to build a separate, more accurately typed specialized version of the function for each type of actual parameter.

The FL type inferencer of Aiken and Murphy [84, 3] treats types as sets of expressions rather than sets of values, avoiding some of the difficulties usually encountered when treating function types. To some degree, our specializer uses similar techniques, specializing functions at each call site in order to compute their return types, instead of attempting to build and instantiate type signatures for functions.

4.6.4 Compilers

The optimizing compiler technology of Chambers and Ungar [19] is very similar to program specialization, except that it occurs at runtime. In particular, their iterative type analysis for loops serves the same purpose as the generalization operation which is often used to compute parameter types/values in online specializers [115, 100, 112]. Chambers' algorithm

pre-abstracts concrete values (such as `1` and `'foo`) to their types (the class of integers and the class of symbols) before entering a loop, thus achieving faster convergence with a possible cost in accuracy (for instance, it is not possible to determine that a particular variable always contains a particular value during a loop). This is acceptable because the compiler is primarily interested in optimizing method dispatch (which depends only on the classes (types) of objects), rather than on performing primitive operations (which require values) at compilation time. The splitting operation, in which portions of the control flow graph following a conditional is compiled separately for each outcome of the conditional, is isomorphic to specializing an `if` expression which has been transformed into continuation-passing style.

4.7 Summary

Although information loss during specialization is inevitable, we have shown how existing specialization systems lose information unnecessarily when computing approximations to return values of residual calls to specializations. The overly general approximations presently used in this case adversely affect the quality of residual programs.

We have presented *return value analysis*, a fixpoint iteration method for computing return value approximations, and have shown how it is implemented in our specializer, FUSE. Adding this methods to FUSE has allowed it to build better specializations of several real-world programs. Indeed, this mechanism is what enabled the construction of an efficient program generator from an online specializer without the use of binding time approximations (*c.f.* Chapter 7 and [95]).

In the future, we plan to explore several avenues. We hope to add support for disjoint union and recursive datatypes to FUSE, increasing the accuracy of generalization, and thus specialization. We may be able to improve the speed of our specializer via static analysis; performing type inference prior to specialization time would establish conservative upper bounds on return value approximations. Such bounds would allow the specializer to halt fixpoint iteration when it detects that the bound has been reached, avoiding unnecessary work.

Chapter 5

Accuracy 2: Parameter Values

The treatment of information recovery in Chapter 4 and in [116] is specific to first-order source programs (actually to first-order residual programs). In this section, we treat an additional source of unnecessary information loss that appears when higher-order programs are considered.

We begin (Section 5.1) by examining what it means to specialize a higher-order program, and reconsider the sources of information loss during specialization of such programs (Section 5.2). We show that the usual approach to specializing higher-order procedures (as used in [115, 11, 23]) requires building specializations that are sufficiently general to be applicable at all of their call sites. Existing methods simply use the overly general approximation "any value" for all parameters to such specializations, even when the actual values can be shown to be more specific, and those more specific values are representable in the specializer's type system. In Sections 5.3 and 5.4, we describe *parameter value analysis*, an algorithm for computing more accurate approximations to parameter values, and its implementation in FUSE. Section 5.5 revisits, in CPS-converted form, the example programs of Section 4.2, and demonstrates how our algorithm achieves the desired specializations. We conclude with a discussion of related work.

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

Figure 5.1: A continuation-passing style length function

5.1 Basics

We begin by considering what it means to specialize a higher-order program. Higher-order programs require that the program specializer be able to operate on first-class functions[1], which may be passed as arguments, returned from procedure calls, and stored in structures. From the specializer's point of view, many of these uses of functions involve little additional difficulty: the specializer explicitly represents closures, and passes, returns, and stores those representations. When a call head evaluates to a closure, the specializer's choices are the same as before: unfold or specialize. The only complication is that lexical access to unknown closed-over variables in the body of the unfolding or specialization must be established, either by arity raising the enclosing specialization (as in Similix-2 [11] and in some versions of FUSE, such as the one described in Chapter 2) or by nesting specializations (as in some versions of the FUSE code generator [115]). As with top-level procedures, this unfolding/specialization process is polyvariant.

Things become interesting when a call head evaluates to an unknown value at specialization time; *i.e.,* when the residual program is higher-order. This situation forces the specializer to residualize the construction of (*i.e.,* build specializations of) all closures which might reach this site at runtime. For example, when the program of Figure 5.1, is specialized on an initial continuation `k` and an unknown list `lst`, both invocations of `k` are reached both by the

[1] As in Chapter 3, we assume, for simplicity's sake, that first-class procedures are distinguished syntactically: top-level (non-first-class) procedures are built by `define` (actually, by the top-level `letrec` form, but `define` is easier to read—clearly, we can always collect the definitions into a single `letrec`), and need not be represented explicitly at specialization time (*i.e.,* the specializer just looks up procedure names in the source program), while first-class procedures are built by `lambda`, and are represented by closures at specialization time. This distinction is also used by Similix-2 [11] and Schism [23]. Our approach does not rely on any such distinction; we make it only to simplify the discussion.

initial continuation[2] , and by the recursive continuation `(lambda (ans) (k (+ 1 ans)))`. This situation forces the specializer to residualize the construction of (*i.e.,* build specializations of) all closures which might reach these sites at runtime. Unlike the case of top-level procedures, in which the specializer can residualize a call simply by building a call to the appropriate specialization, *both* residual calls to `k` must be able to invoke either of *two* specializations at runtime, depending on which `lambda` expression generated the closure which `k` evaluates to. There are several approaches to specializing such programs.

One approach eliminates the problem entirely via environment conversion. The idea is to transform the source program[3], replacing `lambda` expressions with environment constructors (*i.e.,* build a structure representing the `lambda` expression's free variables, along with a tag to indicate which `lambda` expression they belong to), and replace every call site with a `case` dispatch that chooses among the bodies of all `lambda` expressions whose closures that might reach that site.[4] This approach allows a high degree of specialization, since each `lambda` expression may be specialized differently per call site, leading to a cartesian product of specializations (`lambda` expressions $\times$ `call` expressions). We view this transformation as overly low-level because it forces a user-level representation of a virtual machine object, the environment. Such transformations may be counterproductive because they limit the Scheme compiler's ability to choose efficient machine-level representations.[5] Once first-class environments become part of the Scheme language, this approach may become more appropriate. Schooler's IBL [101] system, which is intended to be embedded within a compiler, performs environment conversion of a sort, replacing identifiers with offsets into environment frames; however, it only unfolds `lambda` expressions and cannot build shared specializations.

[2]We use the term "continuation" to refer to a first-class procedure introduced by the CPS transformation, not to any implicit continuation present in the Scheme evaluator or reified by the primitive `call-with-current-continuation`.

[3]This transformation could also be performed during specialization, or in a postpass, but we find it easiest to think of as a source-to-source transformation.

[4]A practical implementation of this might build a top-level procedure for each `lambda`-body, parameterized by the both the free and bound variables. The `case`-dispatch at each call site would apply one of these procedures to the argument values and to the environment. Polyvariant specialization of such a program could potentially build a specialization of each "body" procedure for each non-recursive call site.

[5]For example, once the specializer has mapped closure creation into the creation of a tuple of free variables, it is unlikely that the compiler will be able to share structure between these tuples, as it could with environments. If the same variable is "closed over" several times, this could cause unnecessary copying. Similarly, an explicit representation of environments as tuples may prevent optimizations that map all or part of the environment to registers or stack frames.

Another approach achieves specialization on a per-call-site basis without the need for explicit environments by building all of the specializations at the site of the original `lambda` expression, and choosing among them at runtime. This can be accomplished by allowing the specializer to build as many specializations of each closure as necessary to accommodate all of the closure's call sites. Then, the residual program is transformed as follows. All first-class procedures in the residual program are represented as tuples, with one entry per first-class `call` expression (*i.e.,* `call` expressions whose head must be evaluated at runtime) in the program. Each `lambda` expression is replaced by a tuple constructor whose arguments are either specialized versions of the source `lambda` expression (for positions reaching call sites of the `lambda`), or some placeholder. Each residual call head is replaced by the appropriate tuple accessor. This approach allows a high degree of specialization, but has costs. The specializer pays the cost of building potentially very large number of specializations, many of which may never be invoked, and generates a large residual program. Runtime costs include the constructing and accessing tuples, and creating closures for all of the specialized `lambda` expressions (this is an expense in systems which copy the values of free variables to a closure data structure at closure creation time). We have chosen not to take this approach because we believe that, for many higher-order programs, the additional degree of specialization is not worth this cost; we don't want to risk having a *slower* residual program. To the best of our knowledge, this approach has not been implemented for specialization, though a similar approach has been implemented for binding time analysis [98].

A third approach allows the building of only one residual version of each specialization-time closure object; each specialization-time closure used in a first-class manner is represented by a single `lambda` expression in the residual program (in cases where unfolding or specialization of top-level procedures duplicates `lambda` expressions, each copy may induce a separate specialization). As a consequence of this *monovariance*, if a single closure might reach several such sites, its specialization must be sufficiently general to be applicable at all of them.

(Note that this is monovariance with respect to *specialization-time closures*, which is not as restrictive as monovariance with respect to *source* `lambda` *expressions*. Polyvariance is still available in several forms. First, although a closure may be specialized only once, it can be unfolded (inlined) any number of times. This handles many "packaging" uses of first-class functions, such as passing a closure to `map`. Second, top-level procedures may be both unfolded and specialized an arbitrary number of times, which may result in the duplication

of `lambda` expressions contained in procedure bodies. Thus, a single `lambda` expression will indeed be specialized for different values of its free variables; the limitation is that each of these variants (specialization-time closure) may be specialized for only one argument vector, which must satisfy all of its call sites. This approach is used by FUSE [115], Similix-2 [11], and Schism [23], and is the approach we will consider in the remainder of this chapter.)

5.2 Sources of Information Loss

Now that we have decided how to specialize closures, we reexamine the problem of information loss. As with first-order programs, a specializer will lose information as it processes a higher-order program. All three of the sources we noted for first-order program specialization (*c.f.* Section 4.1) also apply in the higher-order case. These losses stem from (1) the use of a finite type system, (2) the need to generalize when computing approximations to return values of `if` expressions and argument specifications to specializations, and (3) the need to compute approximations to the return values of calls to specialized procedures. With higher-order programs, there is an additional opportunity for information loss: the computation of overly general argument specifications for specializations of first-class procedures.

Although the losses due to the type system and generalization of argument specifications for loop creation are unavoidable, some of the other losses can be avoided rather easily. In particular, the need to compute approximate return values for residual `if` expressions and for residual calls to specialized procedures can be eliminated entirely by transforming the program into continuation-passing style (CPS) [108] form. In CPS programs, all calls are tail calls in the sense that no reductions are performed on their return values; if the initial continuation is the identity function, then every call returns the program's final value. Thus, the specializer can safely use any approximation to the return values of of `if` expressions and residual procedure calls, since nothing is ever going to look at them. Since this is the case, we need not concern ourselves with the problem of computing return approximations in higher-order programs: for the remainder of this chapter, we will assume that all higher-order programs to be specialized have been CPS converted.

This application of the CPS transformation to program specialization was first investigated by Consel and Danvy [26], who used it to improve the accuracy of an offline program specializer lacking the ability to compute accurate return values for `if` statements and

residual procedure calls. CPS overcomes some of these limitations; consider the program fragment

```
(if x
    (cons 1 y)
    (cons 2 z))
```

where `x`, `y` and `z` are unknown. In a specializer like that of Consel and Danvy, which cannot perform generalization, the return value of this fragment is $\top_{Val}$. The best result possible from a specializer with generalization would be "a pair, whose `car` is either `1` or `2`, and whose `cdr` is $\top_{Val}$." This solution is better than $\top_{Val}$, but still loses information, since any reduction depending on the return value can't know if the `car` is `1` or `2` until runtime. When the CPS transformation[6] is applied, the fragment becomes

```
(if x
    (k (cons 1 y))
    (k (cons 2 z)))
```

where `k` is bound to `(lambda (if-result) ... if-result ...)`. In this case, the continuation bound to `k` can be unfolded at each of its call sites; one invocation will be unfolded on `(1 .` $\top_{Val}$`)`, while the other will be unfolded on `(2 .` $\top_{Val}$`)`. This yields an even more accurate specialization than the generalization approach, because, unlike before, the values `1` and `2` are available for performing reductions in the continuation at specialization time. Of course, there are risks: because procedures are specialized with respect to particular continuations, there is less opportunity for sharing (residual programs may become very large), while code duplication may result if continuations are unfolded multiple times on identical inputs (*c.f.* Chapter 2). However, experiments to date [26, 71] suggest that this approach works well, at least under offline methods.

The CPS transformation can also improve the accuracy of specializers that do not compute return values of residual procedure calls. For instance, the tail-recursive length function

```
(define (tail-length lst ans)
  (if (null? lst)
      ans
      (tail-length (cdr lst) (+ 1 ans))))
```

[6] This is not the full CPS transform of [108], which also transforms primitives to take a continuation argument; we need only transform user function definitions and their call sites. Because most specializers have no difficultly computing return values of residual primitive calls, expressions like `(k (cons (car x) (cdr y)))` are considered perfectly acceptable.

specialized on `lst` unknown and `ans` an unknown integer[7], returns $\top_{Integer}$ under specializers which compute return values, but only $\top_{Val}$ under specializers which don't. If we transform it to

```
(define (tail-length k lst ans)
  (if (null? lst)
      (k ans)
      (tail-length k (cdr lst) (+ 1 ans))))
```

we no longer have this problem. When we specialize this transformed version on `lst` unknown, `ans` an unknown integer, and `k=(lambda (loop-result) ...)`, `k` will be applied to (and unfolded on) the approximation $\top_{Integer}$ which is the result we desire. Indeed, this is the case for all first-order programs whose residualization is tail recursive, since, in their CPS-transformed form, recursive calls will pass the same continuation passed to the original call, and all continuation invocations will invoke only one continuation, which can be inlined. As we shall see in Section 5.5, this means that the CPS transformed version of the MP+ interpreter, specialized on the example program of Section 4.2.1, will be able to preserve the shape of the store across the tail-recursive residual loop used to implement the MP+ `while` construct.

Now let us consider the truly recursive formulation of `length` used in Section 4.3. After CPS transformation, it looks like the code shown in Figure 5.1. We will specialize this function on a known initial continuation `k` and an unknown list `lst`. Recall that specializations of closures reaching multiple residual call sites (in this case, both the initial and recursive continuations) must be sufficiently general to be applicable at all of them. *Existing specializers provide such applicability by building all first-class specializations on completely unknown argument values.* This approach builds specializations which are overly general since the call sites may have some amount of information in common, which is not used in building the specialization. In the case of `length`, both the initial continuation (bound to `k`) and the recursive continuation (`(lambda (ans) ...)`) will be specialized on an actual parameter of $\top_{Val}$, forcing the `+` in `(+ 1 ans)` to be left residual in its most general form. Since both residual invocations of `k` will pass an integer argument, it would be preferable to specialize the continuations on $\top_{Integer}$, which would allow the expression `(+ 1 ans)` in

[7] A user would most likely specialize this procedure on `ans=0`, but termination criteria (either manual or automatic) will raise `ans`'s approximation to $\top_{Integer}$ in order to build a finite number of specializations of `length`.

the recursive continuation to be simplified to `(integer+ 1 ans)`.

Similarly (*c.f.* Section 5.5), specializing a CPS transformed version of the MP+ interpreter on a program containing recursive MP+ procedure calls will result in the loss of the shape of the interpreter's store across the residual loop used to implement those procedure calls.

This unnecessary loss of information means that, with respect to the computation of accurate return value approximations, the utility of the CPS transformation is limited to programs whose (first-order) residualization is tail recursive, because all continuations to truly recursive procedures will be specialized on $\top_{Val}$ instead of being unfolded on more specific arguments.

To the best of our knowledge, no existing specializer or transformation method solves the problem of overly general parameter approximations. Much of the existing work on specialization, and experimentation with the CPS transformation, has focused on interpreters for languages with `while` loops, but without procedure call. Since the residual versions of such interpreters are usually tail-recursive[8], the costs of using the overly general approximation "a value" to the actual parameters of residual `lambda` expressions have not been detected. Also, most existing work on higher-order specialization has occurred in the context of offline specialization, which makes the computation of good approximations to parameter values more difficult (*c.f.* Chapter 2).

5.3 An Accurate Specialization Algorithm

For each first-class specialization (residual `lambda` expression), we want to compute an accurate approximation to the parameter values that will be passed to (closures constructed from) it at runtime. For each parameter, this requires finding a single approximation which dominates, in the type lattice, the approximations to the corresponding parameter at each residual call to the specialization. At first, this might seem quite simple: find all residual call sites of a specialization, compute the least upper bound of the approximations to their parameters, and use that approximation to construct the specialization. Two factors complicate this task:

[8] The interpreters contain truly recursive code for interpreting expressions, etc, but all of those loops are fully unfolded at specialization time, leaving only tail recursive loops in the residual interpreter.

1. To obtain an accurate result, the control flow analysis used to find a specialization's call sites must be performed on the *residual* program, which is still being constructed at the time the results of the analysis are required (*e.g.,* in the length program of Figure 5.1, the call to `k` inside the recursive continuation must be discovered before that continuation is specialized.).

2. Computing accurate control flow information, even on a complete program, can be expensive; worse yet, our solution to analyzing an incomplete program will require this control flow analysis to be performed several times.

We address each of these issues in turn.

5.3.1 The Iterative Algorithm

For each first-class procedure to be specialized, we would like to compute an argument vector that is sufficiently general to approximate all values that might be passed at runtime, but which is not overly general. We can do this by generalizing the argument approximations from all of the specialization's call sites; our problem is that, at the time the argument vector is needed, not all of the call sites will have been constructed, since some of them may lie in the body of the specialization itself. This suggests the use of an iterative solution technique, in which we construct an initial specialization based on argument approximations from call sites outside the specialization, then revise the specialization as more call sites are discovered.

One possible algorithm (Figure 5.2) works as follows. Associate two values with each residual `lambda` expression: (1) the closure which was specialized to produce this expression, and (2) the argument vector on which the closure was specialized. Specialize the program as usual, but when a residual `lambda` expression is initially constructed from a closure, do not compute the body by specializing the closure on a completely unknown argument vector. Instead, set the expression's closure field to the appropriate closure, set the argument vector to $\perp_{Val}$, and leave the body empty.

Once specialization is complete, the residual program will contain some number of residual `lambda` expressions. Perform a control flow analysis to find (a conservative approximation to) each `lambda` expression's call sites (*i.e.,* compute a relation `SITES(L)` for each `lambda` expression `L`).[9] For each `lambda` expression, compute the least upper bound of the

[9]If closures constructed from the `lambda` expression can be returned out of the top-level invocation of

```
Specialize program on inputs as usual
Each time we build a residual lambda expression L
     from closure C:
  set L.CLOSURE:=C
  set L.BODY:=empty
  set L.ARGS:=bottom

LOOP:
  Perform control flow analysis on the residual program
  (i.e., compute SITES(L) for all L)
  For each L in the residual program
    Compute A:=LUB(L.ARGS,LUB(S.ARGS)) for all S in SITES(L)
    If A != L.ARGS then
      set L.ARGS:=A
      set L.BODY:=specialize(L.CLOSURE,L.ARGS)
  If any L.ARGS were changed then
    goto LOOP
```

Figure 5.2: An iterative specialization algorithm

current argument vector and the argument approximations at all of its call sites. If this approximation is the same as the expression's argument vector, do nothing.[10] Otherwise, set the expression's argument vector to the new approximation, and re-specialize the expression's closure on the new argument vector. If any residual `lambda` expressions were re-specialized, repeat the process starting with the control flow analysis.

An Example

Consider the `length` function, specialized on a known initial continuation `k=(lambda (result) (+ 5 result))` and an unknown list `x=`$\top_{Val}$. This process is shown in Figures 5.3 and 5.4.

---

the program, the control flow analysis must add a "virtual" call site with completely unknown argument approximations to account for the fact that closures constructed from the `lambda` expression might be invoked on arbitrary values at runtime. This is a standard issue in control flow analysis [105].

[10] It might seem sufficient to halt when the set of call sites remains the same across iterations. This fails because call sites are compared using the identity of `call` expressions in the residual program, and, since specializations are rebuilt on each iteration, any call sites within a rebuilt specialization are guaranteed to appear different on each iteration, resulting in nontermination. Furthermore, the set of call sites of a particular residual `lambda` expression does not grow monotonically during the analysis; for example, a later specialization constructed on general arguments may contain fewer residual calls than an earlier one built on more specific arguments, because loop unfolding in the more specific case may duplicate some call sites.

Initial Program

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) (k (+ 1 ans)))
              (cdr x))))

(define (length2 x)
  (length (lambda (result) (+ 5 result))
          x))
```

After 1 iteration

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) ; specialized on ⊥Val
                <empty>)
              (cdr x))))

(define (length2 x)
  (length (lambda (result) ; specialized on ⊥Val
            <empty>)
          x))

SITES((lambda (ans) ...))    = {(k 0)} ⇒ new ans=0
SITES((lambda (result) ...)) = {(k 0)} ⇒ new result=0
```

After 2 iterations

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) ; specialized on 0
                (k 1))
              (cdr x))))

(define (length2 x)
  (length (lambda (result) ; specialized on 0
            5)
          x))

SITES((lambda (ans) ...))    = {(k 0), (k 1)} ⇒ new ans=⊤Integer
SITES((lambda (result) ...)) = {(k 0), (k 1)} ⇒ new result=⊤Integer
```

Figure 5.3: Applying the iterative algorithm to the `length` program

After 3 iterations

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans)            ; specialized on ⊤Integer
                (k (integer+ 1 ans)))
              (cdr x))))

(define (length2 x)
  (length (lambda (result)             ; specialized on ⊤Integer
            (integer+ 5 result))
          x))

SITES((lambda (ans) ...))    = {(k 0),
                                (k (integer+ 1 ans))} ⇒ new ans=⊤Integer
SITES((lambda (result) ...)) = {(k 0),
                                (k (integer+ 1 ans))} ⇒ new result=⊤Integer
```

Figure 5.4: Applying the iterative algorithm to the `length` program (continued)

On the first iteration of the algorithm, the initial and recursive continuations are both specialized on $\perp_{Val}$, yielding residual `lambda` expressions with empty bodies. Control flow analysis finds one call site for each residual `lambda` expression, namely `(k 0)`. Both continuations are re-specialized on an actual parameter value of `0`, yielding bodies of `(k 1)` and `5`, respectively. This time, control flow analysis finds two call sites for each `lambda` expression, `(k 0)` and `(k 1)`. Once again, we must respecialize, this time on $0 \sqcup 1 = \top_{Integer}$, producing bodies of `(k (integer+ 1 ans))` and `(integer+ 5 result)`. Control flow analysis of this program finds two call sites, `(k 0)` and `(k (integer+ 1 ans))`, for each specialization. This time, the least upper bound $\top_{Integer} \sqcup 0 = \top_{Integer}$ is equal to the argument vector used to build the specializations, so the algorithm terminates.

The final residual program is more specialized than that achieved under standard specialization strategies; if the initial and recursive continuations were specialized on $\top_{Val}$, the applications of `+` in those continuations would not be specialized to `integer+`.

Termination and Correctness

This section informally motivates the termination and correctness of the iterative specialization algorithm. For a more formal treatment, see Section A.5.

The termination of this algorithm depends on two factors: building a finite number of

closures and performing a finite number of respecializations of each closure. The latter is easily achieved; each closure will be respecialized a finite number of times because the argument vector used to respecialize any particular closure is drawn from a finite-height lattice, and rises in that lattice on each subsequent respecialization. The former is more difficult to assure, but is not specific to this algorithm—indeed, it is faced by all existing specializers for higher-order languages. The only way to build an infinite number of closures is to build an infinite number of unfoldings (or top-level specializations) of a loop whose body constructs a closure. Such behavior can be avoided using traditional solutions: limiting unfolding and forcing generalization of certain arguments to specializations [11, 115].

The correctness of this algorithm can be argued inductively. Provided that the control flow analysis is correct (*i.e.,* finds all call sites of each `lambda` expression in the program), the residual `lambda` expressions constructed in iteration k of the algorithm are sufficiently general to be applicable at all call sites in the program produced by iteration $k-1$ of the algorithm. The algorithm terminates when the argument vectors computed from the residual program are the same as those computed from the previous residual program. Because specializations constructed on identical argument vectors are identical, any further iteration would produce an identical program. Thus, if the algorithm terminates at iteration n, the specializations produced by iteration $n+1$ are sufficiently general to be applicable at all call sites in the residual program of iteration n. But since the algorithm terminated at iteration n, the residual programs of iterations n and $n+1$ are the same, and thus the specializations in the program of iteration n are sufficiently general for the call sites in the program of iteration n.

5.3.2 Control Flow Analysis

As it stands, the algorithm of Section 5.3.1 is not practical, for two reasons. First, we have not specified how to compute the necessary control flow information (*i.e.,* the `SITES` relation); many strategies are possible. Second, the algorithm requires that the `SITES` relation be recomputed on each iteration after respecialization has been performed; this can be quite expensive. In this section, we address both of these issues.

Choosing a CFA Strategy

Finding an accurate approximation to the set of call sites reached by a `lambda` expression can be expensive; because the relationship between `lambda` and `call` expressions is data

dependent, it is both a dataflow and a control flow problem. This problem has been treated in detail by Shivers [105] and Harrison [50], while simpler, less accurate solutions are used by Sestoft's "closure analysis" [104], Bondorf's variant of this analysis [11], and Consel's higher-order binding time analysis [23].

All of these analyses compute correct solutions; our concern is with accuracy. If an overly large set of potential call sites is determined for a `lambda` expression, the approximation computed for the `lambda`'s parameters may be overly general. Shivers [105] popularized a taxonomy of analyses (originally due to Hudak [58]) in terms of the depth of the call history used to distinguish between control paths. "0CFA" maintains one set of abstract closures[11], "1CFA" maintains a set of sets, indexed by call sites of the call site's enclosing `lambda`, "2CFA" indexes based on two levels of call sites, and so on. Analyses higher in the taxonomy compute more accurate estimates, but are more costly to compute [68].

0CFA is relatively simple to compute, but unfortunately provides overly general results for continuation-passing-style programs. For example, given the program fragment

```
(define (foo k x)
  (k x))

(cons (foo (lambda (a) ...) 4)
      (foo (lambda (b) ...) 'bar))
```

0CFA will determine that the call site `(k x)` could invoke either `(lambda (a) ...)`[12] or `(lambda (b) ...)` on either `4` or `'bar`, while 1CFA will determine that `(lambda (a) ...)` is invoked only on `4` and that `(lambda (b) ...)` is invoked only on `'bar`. The additional accuracy of 1CFA would thus allow us to specialize `(lambda (a) ...)` on `4` instead of on $\top_{Val}$, which might lead to a significantly better specialization.

Thus, it might appear necessary to use an expensive analysis like 1CFA; luckily, this is not the case. In the example above, 0CFA computes an inaccurate result because it fails to analyze the body of `foo` separately for each of the two calls to `foo`; 1CFA succeeds by keeping additional context to distinguish the calls. Such context is often unnecessary when analyzing *residual* programs because a polyvariant specializer will build different code for different call contexts, which will then be analyzed separately even by 0CFA.

[11] An abstract closure is a `lambda` expression plus an approximate representation of an environment; thus, abstract closures are very similar to specializers' representations of closures.

[12] To be accurate, we mean "closures constructed from `(lambda (a) ...)`;" we omit this phrase for brevity since the meaning should be clear.

First, any `call`s for which the specializer can prove that the head is reached only by closures generated by a single `lambda` expression are either unfolded or specialized; closures need by constructed at runtime, and no further analysis is required. In the example above, if both calls to `foo` were unfolded or specialized on their arguments, `k` would evaluate to a closure, which could then be unfolded or specialized at its call sites, allowing us to take advantage of the values `4` and `'bar`.

Second, even in cases where the specializer constructs a residual call that is reached by several closures, the polyvariant nature of the specializer helps avoid undesirable merging of control paths. Consider the program below, which computes the sum of the values from `0` to `x` where `x` is either a number or a list representing a number in unary notation:

```
(define (sum k x)
  (cond ((number? x)
         (if (= x 0)
             (k 0)
             (sum (lambda (num-ans) (+ x num-ans)) (- x 1))))
        ((list? x)
         (if (null? x)
             (k '())
             (sum (lambda (list-ans) (append x list-ans) (cdr x)))))))
```

and a program fragment containing two calls to `sum`:

```
(cons (sum (lambda (a) ...) x)
      (sum (lambda (b) ...) y))
```

where `x`=$\top_{Integer}$, and `y`=$\top_{List}$. 0CFA on the source program would determine that either continuation could be invoked on either an integer or a list. To terminate, the specializer cannot make use of the value of `k` in building specializations of `foo`, since each iteration builds a new continuation; instead, it may only use some more general value such as "any function."[13] However, the specializer can make use of the value of `x`; because `x`'s type is different at the two call sites, the specializer will build two specializations of `foo`. When 0CFA is run on the residual program, it will notice that the continuation `(lambda (a) ...)` is called only from the specialization with `x`=$\top_{Integer}$, and the continuation `(lambda (b) ...)` is called only from the specialization with `x`=$\top_{List}$, and will compute the approximations we

[13] It could use something like "any closure built from either `(lambda (a) ...)` or `(lambda (num-ans) ...)`," which would distinguish the specializations built by the two invocations, but we will see in a moment that this could be counterproductive.

want. If both call sites had passed "any integer" for `x`, then they would share the same specialization, causing 0CFA to conflate the two closures. Note, however, that such conflation would cause no harm, because both closures would be applied to the same value ($\top_{Val}$). If we had built different specializations for the two call sites merely because different initial continuations were passed as arguments, we would have built duplicate code needlessly, since the value of the continuation is not used in computing the body of the specialization.

Another way to think of this is that 1CFA, 2CFA, etc. split control paths to some fixed depth to obtain more accurate results. A program specializer splits control paths to an arbitrary depth based on the equality of approximations to arguments (*i.e.,* a new specialization is built every time a function is called on a new argument vector, with the expectation that the different information will lead to different reductions). If 0CFA is performed on the residual program, it may needlessly conflate the applications of different closures (*i.e.,* it may erroneously deduce that `(lambda (a) ...)` is called on an argument that really only reaches `(lambda (b) ...)`, and vice versa), but it doesn't matter because this will only happen in cases where those arguments have the same type (otherwise the procedure containing the application would have been split into two specializations), in which case conflating the two applications will do no harm.

Of course, even when analyzing residual programs, there are cases in which a more complex control flow analysis could get better results. For example, our specializer is monovariant over specialization of first-class functions, and thus will fail to build separate specializations for control paths that might be considered separately by 1CFA or other more sophisticated CFA schemes. We are merely arguing that there will be fewer such cases in residual programs than in general programs, making the use of such analyses on residual programs less advantageous than on general programs. Thus, our solution will be based on 0CFA, which is fairly simple and computationally efficient.

Making CFA Efficient

The other problem with our specialization algorithm is one of efficiency: it performs a control flow analysis of the entire residual program on each iteration, even though most of the program doesn't change from iteration to iteration (only the particular specialization(s) being iteratively recomputed will change). Because the respecialization process can both add and remove call sites from a residual `lambda` expression (*c.f.* iterations 2 and 3 in Figure 5.3), each time we perform the control flow analysis, we must restart the abstract

interpretation at "square one," with each `lambda` expression having no call sites. If we were to simply restart the control flow analysis on the existing approximations after removing a call site, the results would be inaccurate because the removed call site would still appear in the result of the analysis. The argument vector of such a site might "pollute" the new argument vector computed by taking the least upper bound of the argument approximations at the various call sites.

We can make two useful observations here. First, simply restarting the control flow analysis on the current call site approximations is never incorrect, merely less accurate. After all, even the approximation "all `lambda`s reach all `call`s of equivalent arity" is correct; it's just not very useful. Thus, we could save time by restarting the abstract interpretation for control flow analysis on the current `SITES` relation after each respecialization phase.

Second, and, for our purposes, more interestingly, *the accuracy loss does not affect the quality of specialization.* To guarantee monotonicity, our algorithm computes the new argument vector by computing the least upper bound of all the new call sites *and the previous argument vector.*[14] Thus, the argument vectors at any "old" call sites not returned by the control flow analysis on this iteration are nonetheless reflected in the new argument vector, meaning that if a less accurate control flow analysis were to mistakenly return an "old" call site, the new argument vector wouldn't be affected. It is safe to restart the abstract interpretation for control flow analysis for any iteration of the specialization algorithm on the approximations computed by the previous iteration, instead of on the bottom element of the abstract interpretation domain. Only the new specializations, and any code called from those new specializations, will need to be re-analyzed.

The observations above suggest the use of an incremental control flow algorithm that, on each iteration of the specialization algorithm, only propagates new call sites (and new residual `lambda` expressions), instead of starting over and re-propagating all call sites and residual `lambda` expressions. This incremental algorithm will compute `SITES` relations containing call sites which no longer appear in the residual program, but this will not affect the argument vectors (or the specializations) computed by the specialization algorithm. This is what we do in FUSE.

[14] If we didn't do this, respecializing a closure might cause nonmonotonicity. Consider a closure c_1 that constructs another closure c_2 containing the only call site of c_3, and passes it to itself recursively. If, for some reason, we respecialize c_1, we will build a new closure c_2', and will temporarily lose the call to c_3 until c_2' is specialized, causing the calculation of c_3's parameter to be nonmonotonic.

5.4 Implementation

This section describes the implementation of parameter value analysis (*e.g.,* control flow analysis and higher-order specialization) in FUSE. We assume, from here onward, that all user procedures and procedure applications in the input program have been CPS-converted; this not only allows us to get good binding times without the need to compute return value approximations (*c.f.* [26] and Section 2.4.2), but also allows us to simplify the control flow analysis.

5.4.1 CFA

Recall that FUSE represents values at specialization time using *symbolic value* objects, which contain a type approximation and a residual code expression. Control flow analysis in FUSE is implemented by adding two new fields to each symbolic value. The *initial sources* field lists all residual `cons` and `lambda` expressions whose output could be returned by the symbolic value's residual expression at runtime. The *final destinations* field lists all residual `car`, `cdr`, and `call` expressions that could destructure (in the case of pairs) or apply (in the case of closures) data structures returned by the symbolic value's residual expression at runtime. To find all residual call sites of a residual `lambda` expression, we simply examine its final destinations field.

During specialization, we maintain the invariant that every destructor that could be reached by the value of a constructor must appear on the constructor's final destinations list, and that every constructor which might reach a destructor at runtime must appear on the destructor's initial sources list. We do this incrementally, as follows:

1. Every symbolic value whose code field is a residual `cons` or `lambda` expression adds itself to its (initially empty) initial sources list.

2. Every symbolic value whose code field is a residual `car`, `cdr`, or `call` instruction adds itself to the final destinations list of its argument (or call head).

3. Every symbolic value created by generalizing two other symbolic values adds the initial sources of both of those symbolic values to its (initially empty) initial sources list. This occurs when a specialization is re-used at a call site other than the one which caused its construction.

4. Adding a final destination to a symbolic value adds all of its initial sources to the new final destination's initial sources list.

5. Adding an initial source to a symbolic value adds all of its final destinations to the new initial source's final destinations list.

6. Whenever a new initial source is added to the final destination list of a pair destructor (`car`, `cdr`), and that initial source is a pair constructor (`cons`), the initial sources of the corresponding argument of the initial source are added to the initial source list of the destructor.

We state without proof that performing these operations is sufficient to maintain the desired invariant. It might seem as though some operations are missing: in particular, when an initial source which is a `lambda` reaches a final destination which is a `call`, one might expect the initial sources of the `lambda`'s body to be added to the initial sources of the `call`. This is unnecessary because we are treating only CPS programs, in which values returned from residual `call` expressions are unimportant—only the values returned from residual *primitive* expressions are used in performing reductions. Similarly, it might appear that when an initial source which is a `lambda` reaches a final destination which is a `call`, the initial sources of the `call`'s arguments should be added to the initial sources lists of the symbolic values representing the `lambda`'s formal parameters. This is unnecessary because this association should not be made until the `lambda` is specialized, at which point the forwarding will be performed by the generalization operation (rule 3).

5.4.2 Specialization

FUSE uses the initial source and final destination information as follows. Every closure object, in addition to fields containing the formals, body, and environment, contains fields containing an argument vector and a specialized body (symbolic value). Each time an initial source which is a `lambda` reaches a final destination which is a `call`, the argument vector of that `call` is generalized with the argument vector stored in the `lambda`'s associated closure object (the first time through, we just use the one from the call). If the old and new argument vectors are equal, initial source information is propagated from the new argument vector to the old one (because the old one is the one whose symbolic values appear in the specialization). If the old and new vectors are different, the new argument vector is used

to build a new specialization, which is then stored, along with the new argument vector, in the closure object.[15]

5.4.3 Example

Once again, let us return to the `length` example:

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) (k (+ 1 ans))) (cdr x))))
```

specialized on `k=(lambda (result) (+ 5 result))` and `x=`$\top_{Val}$. The specializer first constructs a specialization of `length` on unknown `k` and `x`:

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) <empty>) (cdr x))))
```

along with some links. The symbolic value for `k` has an initial source of `(lambda (ans) ...)` and a final destination of `(k 0)`, while the symbolic value for `x` has a final destination of `(cdr x)`. The symbolic value for `(lambda (ans) ...)` has itself as an initial source, and `(k 0)` as a final destination.

When the residual call `(k 0)` is constructed, the specializer iterates through all of the initial sources of `k` (in this case, just `(lambda (ans) ...)`) and recomputes their argument vectors. In this case, the new argument vector for `(lambda (ans) ...)` is `0`. Respecializing yields

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) (k 1)) (cdr x))))
```

During the respecialization process, a new residual call, `(k 1)` is constructed. The specializer must update the argument vectors of all of `k`'s initial sources; in this case, the

[15] This is not entirely correct, because the specialization procedure must be reentrant. That is, during the course of building a specialization of a closure, the closure may be specialized again on more general arguments. This can be solved either through a queueing scheme to remove the reentrancy (cf Figure A.8, or by allowing reentrant invocations, but only storing the new specialization in the closure if the closure's argument vector has not changed since the specialization was requested.

argument vector of `(lambda (ans) ...)`, formerly `0`, becomes $\top_{Integer}$. This change forces a respecialization, building the code

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) (k (integer+ 1 ans)))
              (cdr x))))
```

Once again, a new residual call, `(k (integer+ 1 ans))`, with argument approximation $\top_{Integer}$, is constructed, and becomes a final destination for `(lambda (ans) ...)`. Computing the least upper bound of the argument vectors of `(lambda (ans) ...)`'s final destinations yields $\top_{Integer}$. Since no change occurred, no respecialization is performed. The specializer resumes its normal operation, building a residual invocation of the specialization of `length` on the initial continuation:

```
(define (length2 x2)
  (length (lambda (result) <empty>) x2))
```

The construction of the residual invocation of `length` causes `x2` to pick up the final destination of `x`, namely `(cdr x)`. Similarly, the final destinations of `k`, `(k 0)` and `(k (integer+ 1 ans))`, are added to `(lambda (result) ...)`, which adds `(lambda (result) ...)` to their initial sources. Because new initial sources have arrived at a call, respecialization may be necessary; `(lambda (result) ...)` is respecialized on `0` and then[16] on $\top_{Integer}$, yielding the final program

```
(define (length k x)
  (if (null? x)
      (k 0)
      (length (lambda (ans) (k (integer+ 1 ans)))
              (cdr x))))

(define (length 2 x2)
  (length (lambda (result) (integer+ 5 result)
                  x2)))
```

[16] FUSE doesn't specify the order in which new initial sources are processed; if the call site `(k (integer+ 1 ans))` is processed first, `(lambda (result) ...)` will be respecialized only once, on $\top_{Integer}$, while if `(k 0)` is processed first, respecialization will be performed on both `0` and $\top_{Integer}$. This suggests "batching" initial source updates so that multiple updates to a `lambda` expression's argument vector are processed before respecialization takes place.

In this simple example, we didn't get to see our mechanism propagating initial source information from the arguments of a `cons` out through a `car` or `cdr` operation. This is important in programs where first-class functions are placed into and accessed from list structure (*e.g.,* an initial environment containing functions in an interpreter, or a task queue in a simulator). We did see some benefit from the algorithm's incremental nature, since the correspondences between `(lambda (ans) ...)` and `(k 0)`, `(lambda (result) ...)` and `(k 0)`, and `x` and `(cdr x)` were only derived once; under a traditional CFA framework, these would have been rederived on each iteration of the respecialization algorithm. Such behavior is more important in larger programs where only a small fraction of the program is re-specialized in any given iteration.

5.4.4 Cost

The initial source and final destination slots add a space cost to symbolic values; they also add a cost to the basic operations of the specializer, since building a residual constructor or destructor, or performing a generalization, may result in several links being added. In cases where such an update finds new a call site of a `lambda` expression, it may even result in the (re)computation of a specialization. We have found that, in practice, the cost of this mechanism is low; since updating links is cheap, and specialization is expensive, we only pay a price when respecialization is performed. Since the vast majority of `lambda` expressions in a typical CPS program are merely unfolded, not specialized, our mechanism costs little when it is not needed. In our tests, we have found that the incremental control flow analysis accounts for 10-15% of total specialization time; for programs where no respecialization is required, our algorithm exacts no other overhead.

More importantly, this mechanism is more efficient than the first-order solution of Section 4.3. This is because the first-order solution constructs return value approximations as part of the process of building specializations; since the fixpoint iteration only stops when the same approximation has been built twice, the solution always rebuilds specializations one more time than necessary (*c.f.* Section 4.4.3). This is particularly troublesome in the presence of tail-recursive loops, where the return value approximation is never used during the construction of the specialization.

The CFA solution does not suffer from this problem. In some sense, it "gets one iteration for free" by not building a specialization of a `lambda` expression until a call site is found, which allows it to start at a value higher in the lattice than "bottom," which is what gets

used in the first-order solution. Because information flows only downward, not upward, a recursive call only causes a new iteration if it adds some new information (such as passing a different continuation, which adds a new initial source to a call site in the specialization's body). In particular, there is no cost for tail-recursive calls, because they pass the same continuation as the original call, and thus can't add any new source/destination links. Without new links, there is no way to cause another iteration.

5.5 Examples Revisited

Our specialization technique improves the quality of specialization of residual first-class procedures. Thus, it provides no benefit for those higher-order programs which, when specialized, contain no residual first-class procedures.

In many programs, all of the specialization-time closures are unfolded. For example, many `lambda` expressions are used for implementing nonrecursive abstractions; a closure passed as the function argument to the `map` procedure will be unfolded as part of the unfolding/specialization of `map`. Similarly, in functional language interpreters implementing environments with closures, if all environment lookups are resolvable at specialization time, no higher-order code will appear in the residual program [11].

In other cases, closures are passed upward to implement mechanisms such as jump tables and method dispatching. In direct-style programs, residual `lambda` expressions must be generated; consider

```
((if (> x 0)
     (lambda (y) (+ x y))
     (lambda (y) (+ (- 0 x) y)))
 z)
```

where `x` is unknown and `z=4`. We would like to know that both closures are applied to `4`, but don't know that because neither one is unfolded. Our methods would determine that `y=4`, and would build the appropriate specializations. A better solution is to recognize this example as a common binding time problem which can be solved by CPS converting the program; when we convert it (informally) to

```
((lambda (k)
   (if (> x 0)
       (k (lambda (y) (+ x y)))
       (k (lambda (y) (+ (- 0 x) y)))))
 (lambda (f) (f z)))
```

the continuation `(lambda (f) (f z))` will be unfolded on both branches of the irreducible `if` expression, and both of the `(lambda (y) ...)` expressions will be unfolded on the value `4`, producing an even better specialization than our method (eliminating the `lambda` expressions entirely instead of just specializing them). Similarly, specializing a CPS-transformed version of the tail-recursive `length` procedure

```
(define (length k lst acc)
  (if (null? lst)
      (k acc)
      (length k (cdr lst) (+ 1 acc))))
```

does not require our method because standard specialization methods correctly compute the type of `acc` ($\top_{Integer}$), then unfold the application of `k` on this type. The CPS approach works well for many programs, including direct-style interpreters for small imperative languages with `while` loops, because, under a well-written interpreter, the residual code generated for a `while` loop is tail recursive.

Where our method shines is when first-class specializations *must* be constructed, even when CPS conversion is used. This occurs when not all continuation applications are unfoldable at specialization time; that is, when specializing a recursive procedure where a (non-unfoldable) recursive call passes a continuation different from that passed to the initial call. For direct-style programs, this is the case whenever the (non-CPS) residualization of the program is not tail recursive. The truly recursive length program of Figure 5.1 is one such example; usual specialization methods will specialize the recursive continuation on $\top_{Val}$ rather than on $\top_{Integer}$.

One might at first believe that the problem could be solved by type inference in a postpass or in the underlying Scheme compiler. Such an inference would deduce that `(lambda (ans) (k (+ 1 ans)))` is always called on an integer, and could replace the general `+` operator with `integer+`. This approach has two problems. First, if some other expression (such as a call to `integer?`) depends on the value of `ans`, it will not be reduced by the postpass/compiler, which is not a specializer, and cannot perform arbitrary reductions, construct specializations, etc. Second, scalar type inferencers are unable to optimize the

program when ad hoc types are used (*i.e.,* determine that a closure should be specialized on the pair (`my-type-tag` . $\top_{Val}$)).

Of course, the `length` example is unrealistic, since many programmers would perform tail-recursion elimination by hand. However, many useful programs are truly recursive: `reduce` over non-associative operators, divide-and-conquer problems, programs using the Y operator, and interpreters for languages with recursive procedure calls, are just a few examples.

In the remainder of this section, we demonstrate our mechanism on CPS-converted forms of the examples of Section 4.2.

5.5.1 Interpreter Example

Consider the CPS converted form of the interpreter in Figure 4.3. If we specialize it on the program

```
(program (pars x)
         (dec y)
         (procs)
         (begin
           (while x
             (begin
               (:= y (cons '1 (cons '1 y)))
               (:= x (cdr x))))
           (:= y (cons '1 y))))
```

an unknown input, and an unknown initial continuation, we get the program shown in Figure 5.5. The shape of the store is preserved across the `while` loop; store accesses/updates in the implementation of the statement `(:= y (cons '1 y))` are open-coded `car` and `cdr` operations.

Note that this residual program contains no first-class `lambda` expressions. Our re-specialization mechanism was never used; a traditional specialization method would have worked equally well. This is true because the residual loop `mp-while1598` is tail recursive; it doesn't add to the continuation on each recursive call. It is possible to write programs which, when specialized, produce residual programs with recursive loops. For instance, the program

```
(letrec
    ((mp-while1598
      (lambda (cont1541 store)
        (if
         (cdr (car store))
         (mp-while1598
          cont1541
          (cons
           (cons 'x (cdr (cdr (car store))))
           (cons (cons 'y (cons '1 (cons '1 (cdr (car (cdr store))))))
                 '())))
         (cont1541
          (cons
           (car store)
           (cons (cons 'y (cons '1 (cdr (car (cdr store))))) '()))))))
     (main1597
      (lambda (cont1468 input)
        (mp-while1598
         cont1468
         (cons (cons 'x (car input)) '((y)))))))
 main1597)
```

Figure 5.5: Result of specializing CPS-transformed MP+ interpreter on multiplication program. Completely static formal and actual parameters have been eliminated, but arity raising has not been performed.

```
(letrec
    ((mp-command1729
      (lambda (cont1680 store)
        (if
         (not (null? (cdr (car store))))
         (if
          (not (null? (cdr (car (cdr store)))))
          (mp-command1729
           (lambda (temp1675)
             (cont1680
              (cons
               (car temp1675)
               (cons
                (car (cdr temp1675))
                (cons
                 (cons 'out (cons '1 (cdr (car (cdr (cdr temp1675))))))
                 '()))))))
           (cons
            (cons 'a (cdr (cdr (car store))))
            (cons (cons 'b (cdr (cdr (car (cdr store))))) '((out)))))
          (cont1680 store))
         (cont1680 store))))
     (main1728
      (lambda (cont1599 input)
        (mp-command1729
         cont1599
         (cons
          (cons 'a (car input))
          (cons (cons 'b (car (cdr input))) '((out))))))))
 main1728)
```

Figure 5.6: Result of specializing CPS-transformed MP+ interpreter on minimum program. Completely static formal and actual parameters have been eliminated, but arity raising has not been performed.

```
(program (pars a b)
         (dec out)
         (procs (loop
                 (if a
                     (if b
                         (begin (:= a (cdr a))
                                (:= b (cdr b))
                                (call loop)
                                (:= out (cons '1 out)))
                         (begin))
                     (begin))))
         (call loop))
```

which computes the minimum of two numbers represented in unary notation, produces the residual program shown in Figure 5.6. In this case, each recursive call to the function `mp-command1729` builds up a new continuation `(lambda (temp1675) ...)` to implement the statement `(:= out (cons '1 out))`. Our mechanism correctly computes an approximation `((a .` $\top_{Val}$`) (b .` $\top_{Val}$`) (out .` $\top_{Val}$`))` to the store passed to this continuation. If traditional methods had been used, the continuation would have been specialized on the approximation $\top_{Val}$, and all store accesses in the continuation would have been residualized as loops, rather than as open-coded `car` and `cdr` instructions.

In this case there is only one store access and one store update in the continuation; in a larger program, the cost would be much greater. Even so, using our algorithm reduced the execution time of the residual program by approximately 40% over the residual program produced by the FUSE of [115], which builds first-class specializations on $\top_{Val}$. An additional 50% reduction was obtained when the arity raising enabled by our algorithm was performed.[17] Computing the specialization took 23% longer than under "vanilla" FUSE. Of the extra time required, 49% was spent recomputing one specialization, while the remainder of the time was spent in the incremental CFA algorithm.

5.5.2 Integration Example

We now return to the integration program shown in Figure 4.6. We would like to show how our mechanism specializes the CPS transformed version of this program. If we

[17] These figures are for running the residual code under interpreted MIT Scheme. When the residual program was compiled, our algorithm produced a comparable speedup, but arity raising actually slowed execution. This was due to differences in relative costs of various operations in interpreted vs. compiled code; speedup factors are always highly dependent on the underlying virtual machine.

```
(letrec
    ((integrate-loop71
      (lambda (cont5672)
        (if
         <unfolded version of good-enough? omitted>
         (cont5672 (tc-* (tc-- rhs lhs)
                         (tc-/ (tc-+ (tc-* lhs lhs) (tc-* rhs rhs)) '2)))
         (integrate-loop71
          (lambda (temp61)
            (integrate-loop71
             (lambda (temp62)
               (cont5672 (tc-+ temp61 temp62))))))))))
  integrate-loop71)
```

Figure 5.7: Result of specializing CPS-transformed integration program. Note that the addition of the subinterval estimates (`temp61` and `temp62`) is performed with a specialized addition operator.

---

specialize the CPS-transformed integration program on an unknown initial continuation, `fcn=(lambda (x) (* x x))`, and `lhs` and `rhs` known to be numbers, we get the specialization shown in Figure 5.7. Both of the recursive continuations (`(lambda (temp61) ...)` and `(lambda (temp62) ...)`) are initially specialized on "any number" because the "base case" call site `(cont5672 (tc-* (tc-- ...))` passes a parameter known to be a number. When the second call site `(cont5672 (tc-+ temp61 ...))` is found, its parameter is also numeric, so no respecialization is necessary. FUSE was able to deduce that the parameter passed to both continuations is numeric, allowing it to generate the specialized code `(tc-+ temp61 temp62)` instead of `(+ temp61 temp62)` without unnecessary effort (modulo the cost of performing control flow analysis, of course). Contrast this with the first-order fixpointing solution shown in Section 4.5.2, in which the body of the specialization is computed twice, once assuming that the recursive calls to `integrate-loop` return `bottom`, and once assuming that they return $\top_{Number}$.

5.6 Related Work

Although no existing specializer uses approximations other than $\top_{Val}$ for the values of formal parameters when building specializations of first-class procedures, a variety of techniques have been used to improve the quality of specialization. A number of such techniques are

described in Section 4.6; in this section, we treat only techniques specifically related to higher-order programs. Techniques for first-order programs are discussed in Section 4.6.

The existing specializers for higher-order untyped languages, Similix-2 [11], Schism [23], Lambda-Mix [45] and FUSE [115] ([46] and [101] treat higher-order languages, but cannot build first-class specializations), build a single specialization per dynamic `lambda` expression, using completely dynamic parameter values.

Control flow analysis for Scheme programs has been investigated extensively in the context of parallelization by Harrison [50] and in the context of program optimization by Shivers [105]. Both of these analyses are capable of computing static information about parameters to first-class functions, but have not, as yet, been integrated with partial evaluators that can make use of such information. Existing applications of control flow analysis to program specialization are Bondorf's application [11] of Sestoft's closure analysis [104], Consel's higher-order binding time analysis [23], and Rytz and Gengler's polyvariant binding time analysis [98], all of which operate on source programs prior to specialization, but do not integrate the analysis into the specializer itself. Our work can be viewed as an efficient integration of existing specialization and CFA techniques.

Consel and Danvy [26] suggested the use of the CPS transformation for binding time improvement; their discussion was primarily motivated by dynamic `if` expressions and didn't treat the problem of specializing continuations bound in dynamically controlled non tail-recursive loops. Our work broadens the scope of programs for which their technique is beneficial.

The polyvariant higher-order binding time analysis of Rytz and Gengler [98] operates by replacing each `lambda` expression in the source program with a vector of variants annotated for different binding time signatures, and selecting the proper variant for each call site from this vector at specialization time. This is a specialization-time analogue of the per-call-site specialization technique we discussed in Section 5.1, which constructs multiple variants at specialization time and selects among them at runtime.

The polymorphic inline caching methods of Hölzle [57] take things one step further. At present, specializers only build variants based on different specialization-time information; polymorphic operations which cannot be reduced to a monomorphic form at specialization time are left in their general form. By replacing such operations with a stub routine and a type cache, one could build specializations based on runtime values, which would (1) give more information to the specializer, and (2) avoid the cost of specializing control paths

which are not exercised at runtime. Such methods may become increasingly important when more complex higher-order programs are specialized.

5.7 Summary

Because existing specializers for higher-order languages use completely unknown argument vectors when specializing first-class functions, they build overly general specializations. Our specialization algorithm computes more accurate argument vectors by using the results of a control flow analysis of the residual program, yielding better specializations. We do this efficiently by computing the control flow information incrementally.

Our incremental CFA technique may have applications outside of specialization. For example, polyvariant static analyses (such as BTA) for higher-order programs might encounter a "re-annotate, then recompute CFA" loop similar to the "respecialize, then recompute CFA" loop in our specializer; we believe this to be the case in [98].

Chapter 6

Avoiding Redundant Specialization

In the previous two chapters, we described methods for inferring more information about runtime values at specialization time, allowing the specializer to perform more reductions, and thus produce faster residual programs. We tacitly assumed that all such information, and the increased polyvariance achieved by using it, would be beneficial.

In this chapter, we show that this is not always the case. We contend that the argument values on which a program point is specialized often contain more information than is actually utilized in the computation of the specialization. That is, *different* argument values can produce the *same* specialized procedure. This behavior hurts performance both at program specialization time (by forcing the partial evaluator to compute specializations needlessly), and at runtime (by building unnecessarily large residual programs, which cause degradations in cache and virtual memory performance).

This chapter has seven sections. The first introduces the problem of redundant specialization with a number of examples. Section 2 develops an informal theory of specialization, and provides a criterion for choosing when to re-use an existing specialization. In Section 3, we describe an approximate version of the re-use criterion and its implementation in FUSE. This is followed by a description of some extensions to the re-use mechanism (Section 4), and discussion of where, during the specialization of everyday programs, our mechanism is useful (Section 5). We conclude with a discussion of related work and directions for future work.

6.1 Introduction

Program specializers operate by symbolically reducing the program on the specification of its inputs. Computations that can be performed, given the information available, are performed; otherwise, *residual* code is generated, delaying the computation until the specialized program is run. Not all performable computations are performed: to guarantee termination of the specializer, and to provide sharing in specialized programs, the specializer performs *folding* [17] operations. Folding is done by recursively specializing certain distinguished[1] parts of the program (*specialization points*), and re-using these specialized subprograms (*specializations*) where appropriate. Most specializers use a strategy called *polyvariant specialization* [16], in which program points are specialized with respect to the known values in their argument specifications, and specializations are re-used when all of the known values in their specifications match exactly. Some variants of this method, such as [49, 101, 116] go further, allowing the building of specializations based on known *types* of otherwise unknown values.

In this chapter, we examine this practice of re-using specializations of program points based on matching argument specifications. We contend that the argument values on which a program point is specialized often contain more information than is actually utilized in the computation of the specialization. That is, *different* argument values can produce the *same* specialized procedure.

Different argument values can produce the same specialization in several ways. Consider specializing[2]

[1] Many program specializers, such as Mix [64] and its descendants, make the choice of these points statically, before specialization takes place, but it is also possible to make this decision dynamically during specialization, as is done in FUSE [115] and Turchin's supercompiler [112].

[2] We realize that, under most specializers, this function (and many of our other examples) would never be specialized, but only unfolded. However, by introducing recursion, we can trivially convert all of our examples to ones which must be specialized in order to guarantee termination of the program specializer. For the case of the `test1` function in this example, consider instead specializing

```
(define (must-specialize a x y z)
  (if (null?  a)
      '()
      (cons (if x (+ 1 z) y)
            (must-specialize (cdr a) x y z))))
```

on `x=#t`, `y=2`, and `a` and `z` unknown.

```
(define (test1 x y z)
  (if x
      (+ 1 z)
      y))
```

on `x=#t`, `y=2`, and `z` unknown; the residual code would be

```
(define (res-test1 z)
  (+ 1 z))
```

This specialization will only be re-used at call sites where `x=#t`, `y=2`, and `z` is unknown. If, in the same program, `test1` is also called with different values for `y`, the specialization `res-test1` will not be re-used; instead, a new, *identical* specialization will be built for each value of `y`, and all of these specializations will appear in the residual program.[3] Ideally, the specializer should deduce that all specializations with `x=#t` and `z` unknown are equivalent; `y` is dead, and thus should not be factor in the decision whether to re-use the specialization `res-test1`.

This behavior occurs in cases other than the "dead formal" case above. A specialization can depend on only part of a structured parameter, such as pair, vector, or closure. Specializing

```
(define (test2 x a)
  (+ (car x) a))
```

where `x` is a pair with a `car` of `1`, and `a` is unknown, yields a specialization that is independent of the value of `cdr x`. A naive specializer would build different specializations for `x=(1 . 2)` and `x=(1 . 3)`. Similarly, a specialization can depend only on the type of an argument, rather than its value; if `a` and `b` are unknown, the specialization of

```
(define (test3 x a b)
  (if (number? x) a b))
```

depends only on whether `(number? x)` is true, not on any specific value of `x`. In other cases, type information about a parameter is not used in building a specialization; specializing

[3] In this particular case, a postprocessor could presumably eliminate the extra definitions, but it is possible to construct examples where a reasonable postprocessor could not. Also, even if a postprocessor could improve the residual code, the time wasted building redundant specializations could not be reclaimed.

```
(define (test4 x)
  (if (null? x)
      0
      (1+ (length (cdr x)))))
```

on a list `x`, of unknown length, yields the same result regardless of any knowledge about the type of the elements of the list. Finally, parameters can interact; specializing

```
(define (test5 x y a)
  (* (+ x y) a))
```

on `x=1`, `y=2`, and `a` unknown yields

```
(define (res-test5 x y a)
  (* 3 a))
```

which is valid for any values of `x` and `y` whose sum is `3`, not just for the specific case of `x=1` and `y=2`.

Thus, different sets of argument values can produce the same specialization; conversely, a specialization can often be safely applied to sets of argument values different from those used to build the specialization. If we define the *domain* of a specialization as the set of argument values for which it returns the same result as the original function, we can state the problem more clearly: *a specialization of a function often has a domain which is larger than the set of argument values denoted by the argument specification used to build the specialization.* We advocate reasoning about the domains of specializations in order to re-use specializations more effectively. Of course, specifying some domains, such as that of `res-test5`, would require a constraint solver for real arithmetic, which is far beyond the scope of the approximate solution we offer.

6.2 A Re-Use Criterion for Specializations

This section describes a criterion for optimal re-use of specializations, and comments on why this criterion is not directly implementable.

6.2.1 Formalisms

In Chapter 2, we defined a *specializer* as a function which takes a function definition and a specification of the arguments to that function, and produces a residual function definition,

or *specialization*. The argument specification restricts the possible values of the actual parameters that will be passed to the function at runtime. Although different specializers use different specification techniques, thus allowing different classes of values to be described, they all share this same general input/output behavior. Recall Definition 1 from Chapter 2:

Definition 1 *(Specializer) Let $\mathcal{L}$ be a language with value domain V and evaluation function $E : \mathcal{L} \times V \to V$. Let S be a set of possible specifications of values in V, and let $C : S \to (\mathcal{PS}\ V)$ be a "concretization" function mapping a specification into the set of values it denotes. A* specializer *is a function $SP : \mathcal{L} \times S \to \mathcal{L}$ mapping a function definition $f \in \mathcal{L}$ and a specification $s \in S$ into a residual function definition $(SP\ f\ s) \in \mathcal{L}$ such that*

$$\forall a \in (C\ s)\,[(E\ f\ a) \neq \perp_V \Rightarrow\ (E\ f\ a) = (E\ (SP\ f\ s)\ a)]\,.$$

While constructing a specialization of a particular function, the specializer will often build other specializations (possibly of the same function) as well. To avoid duplication of work (and to guarantee termination of the specializer), most specializers use some form of caching. The class of *polyvariant* specializers re-uses specializations with respect to identical specifications. When the specializer specializes a function f on some argument specification s, it makes an association between f, s, and the specialized (residual) function r, so that subsequent attempts to specialize f on s (or any specification x satisfying $(C\ x) = (C\ s)$) can simply return r without further computation. This form of re-use is obviously safe, since if the specialization r is valid for all values denoted by s, it is surely valid with respect to some other specification that denotes an identical set of values. We will refer to s as the *index* of the specialization r.

Sometimes, as in the examples above, when a function is specialized, not all of the information in the argument specification is used in computing the specialization. Re-using specializations only when the argument specifications are identical misses opportunities to re-use specializations because the argument specification does not always accurately reflect the set of values over which the specialization is valid. That set is defined as follows:

Definition 2 *(Domain of Specialization) If f is a function definition, and r is the result of specializing it on some argument specification, then the* domain of specialization *of r is a set of values $(DOS\ r) \in (\mathcal{PS}\ V)$ where*

$$(DOS\ r) = \{v \in V \mid (E\ f\ v) \neq \perp_V \Rightarrow (E\ f\ v) = (E\ r\ v)\}.$$

The domain of specialization is the set of argument values for which the specialization and the original function definition return the same result, or for which the original function fails to terminate. As before, we allow the specialization to return any value on argument values which cause the original function definition to fail to terminate. The DOS is useful as a safety criterion:

Definition 3 *(Safe Re-Use) When a specializer is faced with the task of specializing a function f on a specification s, it can* safely re-use *another specialization r of f whenever $(C\ s) \subseteq (DOS\ r)$.*

This is a correctness criterion only, and does not guarantee optimality. Consider specializing `(lambda (x y) (+ x y))` on unknown `x` and `y`, producing `(lambda (x y) (+ x y))`. The correctness criterion indicates that re-using this specialization for the case `x=1, y=2` is safe; however, it does not indicate that not re-using the specialization will allow us to build a "better" one, namely `(lambda (x y) 3)`. The domain of specialization doesn't indicate whether the specializer could generate a "better" specialization for a smaller domain.

Before proceeding, we must codify this notion of "better." Intuitively, the goal of a specializer is to perform reductions at specialization time, so that these reductions will not be performed at runtime. Because these reductions make use of information available at specialization time, that information must also be true of the values passed in at runtime. Thus, each time the specializer makes use of specialization-time information to perform a reduction (and thus build a more efficient specialization), it potentially reduces the domain of the specialization. Making use of this observation, we can define an optimality relation on specializations:

Definition 4 *(Strictly More Specialized) A specialization p of a function f is* strictly more specialized *than another specialization q of f if and only if $(DOS\ p) \subset (DOS\ q)$.*

Thus, for `x=1`, `y=2`, the specialization `(lambda (x y) 3)` is strictly more specialized than the specialization `(lambda (x y) (+ x y))` because it has the domain of pairs of

numbers whose sum is 3, rather than the domain of pairs of numbers.

Now that we have both a safety criterion and an optimality criterion, we can formulate a re-use criterion:

Definition 5 *(Optimal Re-use) Given a function definition f and argument specification s, a specializer may* optimally re-use *a specialization r of f if and only if it is safe (Definition 3) to do so, and $(SP\, f\, s)$ is not strictly more specialized (*i.e., *not a better specialization) (Definition 4) than r. In other words, r can be optimally re-used if and only if*

$$(C\, s) \subseteq (DOS\, r) \wedge \neg\, [(DOS\, (SP\, f\, s)) \subset (DOS\, r)]\,.$$

6.2.2 Practicalities

Given a re-use criterion for specializations, what can we do with it? As it stands, the definition has two major practical deficiencies:

1. The real specializer is a program, not a mathematical function. It cannot manipulate arbitrary, potentially infinite sets of values (such as the domain of specialization or concretizations of argument specifications) directly; instead, it must operate on finite, approximate representations of these sets. We will use elements of a type lattice as representations; this will allow us to recast all of the above definitions using domain operators instead of set operators. Although the accuracy of approximation will depend on our choice of type lattice; the correctness of our algorithms will not depend on any particular choice.

2. The optimality test requires that the specializer compute the specialization $(SP\, f\, s)$ before deciding whether to re-use a pre-existing specialization r of f. While this method produces the desired output, it does not satisfy our goal of saving work in the specializer itself, since the specializer would need to build a specialization in order to determine whether it is necessary to build that same specialization. Since, in some cases, it is possible to prove that $(DOS\, (SP\, f\, s)) = (DOS\, r)$ from r and s alone, without computing $(SP\, f\, s)$, our strategy will be to attempt this proof, and apply the optimal re-use criterion only when the proof fails.

6.3 Re-use of Specializations in FUSE

This section describes the implementation of the above criterion in a first-order version of FUSE; higher-order extensions are described in Section 6.4.2.

6.3.1 Specifying Values with Types

A specializer operates on programs and argument specifications. Because Scheme functions can take multiple arguments, FUSE uses *value specifications* to specify the values of individual arguments; an argument specification is simply a tuple of value specifications. Recall that a value specification is a finite description of a possibly infinite collection of values that might appear in its place at runtime. Because such a description is finite, it is necessarily approximate; to ensure correctness, it must also be conservative. That is, any specification must denote at least those values which will be passed as arguments at runtime; often it will denote more values.

FUSE draws its value specifications from a type system, which varies among implementations of FUSE (*c.f.* Section 3.1.2). For the sake of discussion, we will assume a domain similar to that described on Page 3.1.2, which includes pairs, concrete scalars, and "top" elements of various scalar subdomains, such as $\top_{Number}$ and $\top_{Boolean}$. For example, a pair whose `car` is completely unknown and whose `cdr` is known to be a number is described as $(\top_{Val} \ . \ \top_{Number})$. As we described in Section 3.2.1, these value specifications are embedded inside *symbolic values*, which also include other attributes such as residual code expressions; the value specifications can be retrieved using an appropriate projection function. An *argument specification* is simply a tuple of value specifications.

6.3.2 Computing Approximations of the DOS

Now that we have refreshed our memory of FUSE's argument specifications, we turn our attention to using them to compute the domain of specialization of a particular specialization. As mentioned before, the DOS is often infinite, and cannot be computed directly. Instead, we compute an argument specification which safely approximates the DOS, in the sense that its concretization is a subset of the DOS. We say an argument specification x is *strictly more general* than another specification y if $x \sqsupset y$. When x is strictly more general than y, we also say that x has *strictly less information* than y, or conversely, that y has *strictly more information* than x. For a specialization r of a function definition f, the approximation we

are seeking is the most general argument specification s such that $(SP\ f\ s) = r$. We will call this specification the *Most General Index*, or MGI, of the specialization r. The remainder of this section describes, informally, how FUSE computes the MGI. A more formal treatment is given in Section A.6.

The process of specialization incrementally reduces the domain of the specialized function. The original (unspecialized) function definition has the largest possible domain, since any values could potentially be passed in at runtime. When specializing a function definition, the specializer uses the information in the argument specification to perform a reduction if that reduction is valid for all instances of the specification. Some reductions (such as reducing `(+ 1 2)` to `3`, or reducing `(if #t 'foo 'bar)` to `foo`, where the `1`, `2`, `#t`, `foo`, and `bar` are constants in the program) are always valid, regardless of the specification, and do not reduce the domain of the specialization. Others, however, that rely on information in the specification (such as reducing `(number? x)` to `#t` when `x` is specified as $\top_{Number}$, or reducing `(+ y 1)` to `3` when `y` is specified as `2`) reduce the domain of the specialized function.

Our approach relies on maintaining, at all times, an approximation of the most general index of each function currently being specialized. Just as argument specifications are comprised of value specifications, one per argument, the approximation of the MGI is maintained on a per-argument basis. For each argument to a specialization, the corresponding portion of the approximation starts out at $\top_{Val}$; each time the specializer performs a reduction that reduces the domain of the specialization, it updates the approximation. When the specializer is finished building the specialization, the approximation will be correct, and can be cached along with the specialization and the index.

We implement this in FUSE by adding a new attribute, the *domain specification*, to all symbolic values. Like value specfications, we find it useful to have structured (*e.g.,* pair- and closure-valued) domain specifications contain symbolic values rather than domain specifications. This is necessary so that changes to the domain specification of a symbolic value inside a structure are reflected in the domain specification of the entire structure. We accomplish this by having domain specifications be elements of Val' instead of *Val*. Since it's easy to construct an appropriate projection function that coerces a domain specification to an element of *Val* by taking the domain specifications of all embedded symbolic values, we will treat domain specifications as though they were elements of *Val*.

Whenever FUSE decides to specialize a function on some argument specification, it

makes copies of all of the symbolic values in the specification (by instantiating their value specifications on `VAR` expressions computed from the formal parameter names) , and sets the domain specification of each copy to $\top_{Val}$. As it constructs the body of the specialization (by applying the body to the new, copied argument specification), it uses the value specifications to perform various reductions. Whenever a reduction reduces the DOS of the specialization, the corresponding domain specification is side-effected to a value lower in the lattice (though never lower than the corresponding argument specification; for any symbolic value with value specification s and domain specification d, we require that $s \sqsubseteq d$). This "lowering" can happen in one of three cases:

- *conditionals:* When the test of a conditional is known, the conditional is reduced to one of its two branches, and the domain specification of the test is set to the truth value of the test. If the test is unknown, its domain specification is left unchanged.

- *primitives:* When a primitive is reduced, any information it uses about its argument(s) must be reflected in the domain specifications of those argument(s). That is, if a primitive p is reduced on an argument with value specification s, it must set the argument's domain specification, d, to a value sufficiently low in the lattice that $\forall x \in (C\,d)\,[(E\,(SP\,p\,d)\,x) = (E\,(SP\,p\,s)\,x) = (E\,p\,x)]$. Furthermore, it should never be the case that reducing a primitive on a general argument should use more information than reducing it on a more specific argument; that is, if reducing p on s sets the argument's domain specification to d, and reducing it on s' sets it to d', then $s \sqsubseteq s' \Rightarrow d' \not\sqsubseteq d$. (This treatment assumed a unary primitive; for binary primitives, we just extend the above to tuples of value and domain specifications).

  For instance, executing `number?` on a symbolic value denoting `6` reduces to `#t`; this used the fact that `6` is a number, so its domain specification is lowered to $\top_{Number}$. Executing `1+` on a symbolic value denoting `6` returns `7`; in this case, the `6` was used for its value, and its domain specification is lowered to `6`.

  Note that this strategy occasionally loses accuracy due to the limitations of the FUSE type system. One example is reducing `(null? x)` to `#f` when `x` is known to be the pair `(1 . 2)`. The information being used is that `x` is not the empty list; but there is no way of expressing this in the type system. The best we can do is to find the highest point in the lattice which lies above `x` but not above the empty list;[4] in this case, it

[4]This method presumes that a unique such point exists; this happens to be the case for FUSE's type

is `(pair (`y `.` z`))`, where y and z are symbolic values with value specifications $\top_{Val}$. This gives a conservative estimation of the DOS, because we are restricting it to all pairs rather than all objects other than the empty list. A similar case is reducing `(+ a b)` to `3` when `a` is known to be `1` and `b` is known to be `2`. Ideally, this should restrict the domain to those cases where the sum of `a` and `b` is `3`, but since we cannot express that, we instead restrict the domain to one where `a` is exactly `1` and `b` is exactly `2`; in this case, we do no better than normal polyvariant specialization. A better type system capable of expressing negation would be helpful here.

- *function application:* The specializer must make the usual unfold/specialize decision. If the decision is to unfold, the body is evaluated on the actual parameter values in the usual way; any conditionals, primitives, or function applications in the body will alter the domain specifications of the actual parameters to the unfolded procedure. If the decision is to specialize, the specializer will either re-use a previous specialization, or construct a new one (how this choice is made is described in the next section). After this choice is made, the domain specification of each actual parameter must be lowered to indicate that the information in the arguments was used to choose a particular specialization. We do this by setting the the domain specification of each actual to the greatest lower bound of the actual's domain specification and the corresponding element of the chosen specialization's MGI.

Once a specialization is complete, the domain specifications of the symbolic values in the index of the specialization form the MGI for that specialization. The specializer caches this approximation along with the specialization index and the specialized function body, so that it can be used when making re-use decisions for that specialization.[5]

For an example of MGI computations, see Figure 6.1.

6.3.3 Using the Approximations to Control Re-Use

We have shown how the specializer computes an approximation to the domain of specialization of each specialization it builds; we now turn our attention to using this information

lattice, because $\bot_{Val}$ is the only type with multiple parents. More complicated type lattices would require a different mechanism for expressing the complement of a type.

[5] Actually, the specializer merely stores the vector of symbolic values on which the specialization constructed; taking the value specification projection gives the index, while taking the domain specification projection gives the MGI.

```
;;; This figure is a transcript; lines beginning with "=>" were typed by the user

=> (define test
    '(lambda (x y z)
             (if x
                 (+ 1 z)
                 (* 10 (car y)))))

;;; The results of specialization are shown in a pretty-printed form
;;; which shows the index and MGI of specializations as well as the
;;; residual code.  Abstract types such as ⊤Val and ⊤Number
;;; print as top and top-number.

=> (sp test #t '(2 . 3) (a-value))
(((no-name                          ; name of specialization
   ((#T (2 . 3) top)                ; index (value attributes only)
    (#T top top)))                  ; MGI (valid for all y)
  (lambda (x3 y2 z1) (+ 1 z1))))    ; body of specialization

=> (sp test (a-value) '(2 . 3) (a-value))
(((no-name
   ((top (2 . 3) top)               ; index
    (top (2 . top) top)))           ; MGI (didn't use cdr of y)
  (lambda (x3 y2 z1)                ; body
    (if x3 (+ 1 z1) 20))))

=> (sp test (a-value) `(,(a-number) . 3) (a-value))
(((no-name
   ((top (top-number . 3) top)      ; index
    (top (top-number . top) top)))  ; MGI (did use type of car of y)
  (lambda (x3 y2 z1)                ; body
    (if x3
        (+ 1 z1)
        (tc-* 10 (tc-car y2))))))   ; tc-*   assumes its arguments are numbers
                                    ; tc-car assumes its argument is a pair
```

Figure 6.1: Examples of specializations and their MGIs

to control re-use of specializations.

As stated above (Definition 3), having the DOS allows us to decide when it is safe to re-use a specialization. Since the DOS is not computable, we will use its approximation, the MGI instead, substituting domain operators for the set operators in Definitions 3 and 5. Given a specialization r of a function f, at any particular call site of f on arguments a, if the value specification of a is less than the MGI of r, then it is safe, though not necessarily desirable, to re-use r.

Definition 5 provides one method of determining whether to compute a new specialization or re-use an existing one. It suggests building a new specialization, then comparing the domains (actually, the best we can do is to compare the MGIs, which are approximate specifications of the domains) of the two specializations: if the domain of the new specialization is smaller, use it; otherwise, discard it and re-use the old specialization. This technique works, and FUSE sometimes is forced to use it. Unfortunately, it has the disadvantage that it often computes specializations only to find that they are redundant. This technique will therefore produce better residual programs, but will not save effort at partial evaluation time.

A somewhat more efficient technique exists: we can often prove that the domains of the old and new specializations will be the same without building the new specialization. Assume function f has been specialized on an index (value specification) v, producing specialization r with MGI (domain specification) d. Specializing f on another value specification v' where $v \sqsubseteq v' \sqsubseteq d$ will yield a copy of r. Intuitively, v' contains enough information to allow us to re-use r safely, but no new information which could make building a new specialization worthwhile. Consider the cases:

1. $(v' \not\sqsubseteq d)$. In this case, since the specialization r is guaranteed to be valid only for values denoted by d, and v' may denote values not denoted by d, it is not safe to re-use r. Therefore, build a new specialization on v'.

2. $v \sqsubseteq v' \sqsubseteq d$. In this case, v' denotes a subset of the denotation of d, so it is safe to re-use the specialization. Given v, only the information in d was actually used in building r, and v' contains less information than v, so we have nothing to gain by building a new specialization (which would necessarily have MGI d). Therefore, re-use r.

3. otherwise. In this case, v' denotes a set of values which is not a superset of those denoted by v. Even though it is safe to re-use r, building a specialization on v' might

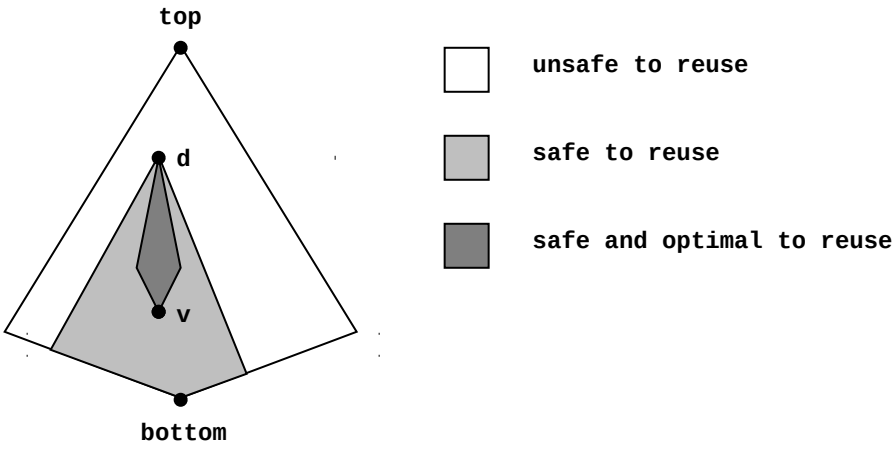


Figure 6.2: Basic re-use criterion. Given an argument specification v', the reusability of an existing specialization with argument specification v can be determined by locating v' in the diagram above. Re-use is safe when $v' \sqsubseteq d$ and is optimal when $v \sqsubseteq v' \sqsubseteq d$.

---

be worthwhile. Therefore, build a new specialization n; if it is more specialized than r (by the criterion of Definition 4), then use it, otherwise discard it and re-use r. In the latter case, discard only the body of the new specialization n; a relation between the index of n (namely, v') and the MGI and body of the old specialization r is still cached. Thus the knowledge that r can be re-used for specification v' is not lost, saving the specializer from performing this needless computation again (we will improve this portion of the criterion in Section 6.4.3).

A diagram of these cases is shown in Figure 6.2. Case 2 is important because it allows the specializer to avoid redundant work. For a more formal treatment of these cases, see Section A.6.

Before presenting FUSE's algorithm for re-using specializations, we must cover one more technical point dealing with recursion in specializations. FUSE specializes functions in a depth-first manner; during the specialization of a function f, it may specialize other functions that f calls. Because Scheme allows recursion, f may call itself. When the specializer reaches a recursive call to f, it has a problem, because the decision whether to re-use the specialization of f currently being built depends on the MGI of this specialization, which hasn't yet been fully computed. The specializer cannot use the partially-computed

```
specialize(fcn, args, cache)
  for each specialization r of fcn in cache
    let the_mgi = if in_progress(r) then index(r) else mgi(r) endif
    if (index(r) <= args) and (args <= the_mgi)
    then return r
    else
      ; n is the cache object representing the new specialization
      let n = <parent=fcn, index=copy of args, mgi=empty, body=empty>
      set domain specifications of all symbolic values in index to top_val
      add n to cache
      build_specialization(n) ; side effects n and cache
      for each specialization r<>n in cache
         if not in_progress(r) and (n.mgi = r.mgi)
         then set n.body = r.body; return r endif
      return n
    endif

build_specialization(n)
   let body = n.parent specialized on n.args
              ; unfolds parent on arguments
              ; makes recursive calls to specialize if necessary
              ; computes mgi(n) in domain specification slots of n.args
   set n.body = body
   set n.mgi = collect domain_specifications of n.args
```

Figure 6.3: FUSE's algorithm for re-using specializations

MGI, because the final MGI might be more restrictive, and thus the recursive call might be an unsafe one (*i.e.* the specialization of f might end up being for a domain that doesn't include the values being passed to the recursive call). Instead, when deciding whether to re-use a specialization that is currently being built, FUSE uses the specialization index (which is guaranteed to be less than or equal to the final MGI) as the MGI. This strategy will occasionally cause the specializer to build a redundant specialization of f, which will then be detected and removed when both specializations of f are complete. Such cases are rare in practice.[6] One might consider postponing the re-use decision rather than using an inaccurate MGI (the argument specification). Katz [70] has proposed such a mechanism (in his case, he needs the more accurate MGI information because he uses it to make the specializer terminate).

FUSE's algorithm for re-using specializations is shown in Figure 6.3. As an example of its usage, consider the function definition

```
(define test
  '(lambda (x y z)
     (letrec
         ((f (lambda (n l)
               (if (null? l)
                   '()
                   (cons
                    (if (number? n)
                        (+ (car l) 1)
                        (car l))
                    (f n (cdr l)))))))
       (let* ((t1 (f x z))
              (t2 (f y z)))
         (cons (t1 t2))))))).
```

Using

```
(sp test 1 (a-number) (a-value))
```

will specialize this function definition on `x=1`, `y` a number, and `z` unknown. The specializer first builds a specialization of `f` for `n` specified as `1`. However, the value `1` is never used, and thus `n`'s entry in the MGI will be $\top_{Number}$. When the specializer specializes the second

[6] Many online specializers, including some versions of FUSE, use a termination criterion which forces all non-unfoldable recursive calls to a function f within a specialization of f to invoke that same specialization, generalizing the arguments as necessary to make this possible. In such cases, using the specialization index when examining incomplete specializations does not risk building a redundant specialization.

call to `f`, where `n` is bound to $\top_{Number}$, it finds that the existing specialization satisfies the re-use criterion, and reuses it. The residual program is thus

```
(((no-name                                ; name
   ((1 top-number top)                    ; index
    (top-number top-number top)))         ; MGI
  (lambda (x3 y2 z1)                      ; body
    (cons (f6.1 x3 z1) (f6.1 y2 z1))))
  ((f6.1                                  ; name
    ((1 top)                              ; index
     (top-number top)))                   ; MGI
   (lambda (n5 l4)                        ; body
     (if (null? l4) '()
         (cons (+ (car l4) 1)
               (f6.1 n5 (cdr l4)))))))).
```

Running the command

```
(sp test (a-number) 1 (a-value))
```

to invoke the specializer produces a different behavior. In this case, the first specialization built is for `n` specified as $\top_{Number}$. This type information is used, and thus the MGI also specifies `n` as $\top_{Number}$. When the second call to `f` is specialized, the value of `n`, namely `1`, is more specific than the index of the existing specialization, so the re-use criterion fails, and a new specialization is built. However, in this new specialization `n` is used only for its type and not for its value, so the MGI specifies `n` as $\top_{Number}$. At this point, the specializer finds that it has already built a specialization with this same MGI, and re-uses it, discarding the new specialization, giving the following residual program:

```
(((no-name                                ; name
   ((top-number 1 top)                    ; index
    (top-number top-number top)))         ; MGI
  (lambda (x3 y2 z1)                      ; body
    (cons (f6.1 x3 z1) (f6.1 y2 z1))))
  ((f6.1                                  ; name
    ((top-number top)                     ; index
     (top-number top)))                   ; MGI
   (lambda (n5 l4)                        ; body
     (if (null? l4) '()
         (cons (+ (car l4) 1)
               (f6.1 n5 (cdr l4)))))))).
```

Thus, in this case, FUSE's strategy produces the desired residual program, but does not save work at specialization time. Section 6.4.3 describes an improvement which avoids such

redundant work by keeping track of which information was demanded but unavailable; in this case, it would deduce that the numerical value of `n` was not demanded, and would not build and discard the specialization for `n` as `1`.

6.4 Extensions

This section describes three classes of extensions to the basic re-use algorithm given above. The first class of extensions is necessary to make the re-use mechanism compatible with the first-order information preservation mechanisms of Chapter 4 and [116, 97]. The second subsection describes how our mechanism interacts with higher-order specialization as implemented in FUSE, and with the higher-order specialization techniques described in Chapter 5 and [97]. Finally, we describe two extensions which improve the efficiency of specialization by avoiding the construction of specializations which are sure to be discarded, and by limiting the use of information during the construction of specializations. These extensions are, for the most part, orthogonal, and do not interfere with one another.[7]

These extensions will require that we modify the specializer. To aid in understanding the scope of these modifications, we separate them into three classes:

1. Modifications to the re-use criterion of Section 6.3.3. That is, change the inequality involving the arguments, specialization index, and MGI in the `specialize` procedure of Figure 6.3.

2. Modifications to the computation of domain specifications (and thus MGIs), as described in Section 6.3.2. This may involve changes to the implementation of primitives, special forms, and function application in the specializer (*i.e.,* anything invoked by the `build-specialization` procedure of Figure 6.3).

3. Modifications to other operations in the specializer unrelated to specialization re-use. This includes the generalizer as described in [115] as well as the type inference mechanisms of [97].

When we describe modifications, we will categorize them using the above terminology.

[7] One exception is that the first-order and higher-order mechanisms of [97] are mutually exclusive, so their re-use-related extensions need not be compatible.

6.4.1 Handling FUSE Type Inference

Unlike most other specializers, FUSE reasons about the values returned by residual conditional expressions and calls to specialized procedures, using the techniques of generalization and fixpoint iteration, respectively. Each of these techniques requires that we modify the re-use mechanism slightly. These modifications will not affect the re-use criterion, but will affect both the computation of domain specifications and the operation of the type-inferencing portion of the specializer.

Residual Conditionals

FUSE is often able to compute useful information about the values returned from `if`-expressions with unknown tests, by returning the generalization of the value specifications of the two arms of the conditional [115]. For instance, specializing

```
(lambda (p)
  (number?
    (if p
        1
        2)))
```

on completely unknown `p` yields the original function definition (modulo renaming) under most specializers, but instead yields

```
(lambda (p x y z)
  #t)
```

under FUSE.[8] This mechanism for propagating information out of residual code necessitates a minor modification to our procedure for computing the MGI, as described above Section 6.3.2. Consider specializing

```
(lambda (p a b)
  (let ((c (if p a b)))
    (cons c (number? c))))
```

where `p` is unknown, `a=1` and `b=2`. The result of `(if p a b)` (bound to `c`) will be a symbolic value with value specification $\top_{Number}$ and domain specification $\top_{Val}$. Applying

[8] For more realistic examples of the usefulness of such generalization, see [97, 116]

the primitive `number?` will make use of the fact that its input is a number, and will lower the domain specification of (the symbolic value bound to) `c` to $\top_{Number}$. Unfortunately, the domain specifications for (the symbolic values bound to) `a` and `b` were never changed; therefore, the MGI for the the specialization will be $(\top_{Val}\ \top_{Val}\ \top_{Val})$ when it should be $(\top_{Val}\ \top_{Number}\ \top_{Number})$.

There are two possible solutions. First, we can force the generalization operator to "use" as much about its arguments as it provides in its result. In the example above, generalizing `a=1` and `b=2` would return $\top_{Number}$, and set the domain specifications of (the symbolic values bound to) `a` and `b` to be $\top_{Number}$. This "eager" computation of domain specifications produces an overly specific MGI if the result of the generalization is not completely used.

Instead, we compute the domain specifications of inputs to generalizations lazily: whenever the domain specification of a symbolic value obtained from a generalization is lowered, the domain specifications of both inputs to the generalization must also be lowered accordingly. To accomplish this, we add a `parents` attribute to all symbolic values; whenever a generalization is performed, the parents field of the output symbolic value is set to point to both of the input symbolic values. All side effects to the domain specification slot of a symbolic value are propagated, recursively, to its parents. Thus, in the example above, the result of evaluating `(if p a b)` returns a symbolic value s with value specification $\top_{Number}$, domain specification $\top_{Val}$, and parents $(a\ b)$, where a and b are the symbolic values bound to `a` and `b`, respectively. When the `number?` operator lowers the domain specification of its input, s, to $\top_{Number}$, it also lowers the domain specifications of s's parents, a and b, to $\top_{Number}$, and the correct MGI is computed.

This extension is a type 2 modification; it affects only the computation of domain specifications.

Further complications arise because of FUSE's ability to return information out of calls to specialized functions. Most specializers simply treat the output of such a residual function call as though it were $\top_{Val}$. However, FUSE is different; for each specialization it builds, FUSE computes a generalized return value specification, which specifies those values that could be returned from *any* call to this specialized function. This information can then be used in specializing the function containing the residual call The generalized return value specification for a given specialization is computed by assuming that the return value is $\bot_{Val}$, and iteratively recomputing the specialization until the return value converges. For

examples, see Chapter 4 or [97, 116].

This mechanism conflicts with the specialization re-use analysis because the return value specification is computed during specialization, and is based on the specialization's index, not its MGI, and is thus valid only at the original call site, not at subsequent call sites. For example, specializing the function

```
(lambda (x y z)
  (if x
      y
      z))
```

on `x` specified as `#t`, `y` specified as $\top_{Number}$, and `z` specified as $\top_{Val}$ yields

```
(lambda (x y z)
  y)
```

a return value specification of $\top_{Number}$, and a MGI in which `y` is specified as $\top_{Val}$. Thus, we can safely and optimally re-use the specialization on a different set of parameters where `y` has any specification that lies below $\top_{Val}$ but not below $\top_{Number}$. Note, however, that the return value specification is in error; although the specialization is indeed applicable to `y` specified as $\top_{Val}$, the return value specification for that case would be $\top_{Val}$, not $\top_{Number}$. This occurs because the return value specification is built using the *value specifications* of the arguments the function was specialized on, even though the generated code may be valid for more general inputs (up to and including the MGI).

We must choose between computing a single, general, return value specification that is valid for all calls to the specialization, or computing a different one for each call site. Computing a single value is straightforward; we must ensure that only information used in the building of the specialization is used in computing the return value specification; values that are not used may not constrain the return value. This has the disadvantage that accuracy is lost, since all calls to a specialization must share a common return value specification.

Computing a different return value per call site is more attractive, since it preserves more information. However, since FUSE computes return value specifications by computing the body of the specialization and taking its value specification attribute, doing this would seem to require recomputing the body of the specialization on each set of actual parameter specifications. Luckily, we need not perform such a recomputation. At runtime, the value

(or its subparts, if it is a data structure) returned by a call to a user procedure may come from one of three places. A returned value can be:

1. a constant in the procedure (*i.e.,* `4` or `"foo"`), or
2. constructed by the procedure (*i.e.,* the result of `cons`, `+`, or `lambda`), or
3. all or part of one of the procedure's arguments (*i.e.,* `(car <formal>)`).

These are the only possibilities. Constants are not a problem, as they will be the same for every call site. Values constructed by the procedure are also valid for every call site, since computing them alters the domain specifications of their arguments, thus restricting the MGI of the specialization such that only call sites that would construct the same values will re-use this specialization. This leaves the case of argument values that are "passed through," which can be solved by changing the interpretation of the return value specification.

Instead of treating the return value specification as a value, and using it as the return value specification for all call sites, we will instead treat it as a template to be filled in[9] at each call site. When the decision to re-use a specialization is made, the specializer matches the symbolic values in the arguments against those in the specialization index, and builds a substitution environment. Any part of the specialization index appearing in the template (which was built from the specialization index during the specialization process) is replaced by the corresponding part of the argument specification. There is a technical point related to generalization: if a value in the return value approximation was obtained by generalizing two other values, then the generalization must be performed again after substitution has taken place. This requires that the specializer keep track of how generalized values were computed; the `parents` attribute used for handling residual conditionals can be used to do this. If a template has symbolic values in its `parents` attribute, we process them, then generalize the resluting values. Consider specializing the function

```
(lambda (a b c d)
  (list 44 (+ a 1) c (if b c d)))
```

on `a=2`, `b` unknown, `c=(foo 1)` and `d=(foo 2)`. We will use the names a, b, c, and d to denote the symbolic values bound to the formal parameters `a`, `b`, `c`, and `d`, respectively. The specializer builds a specialization

[9] In other areas of computer science, the "filling in" of a template is often called *instantiation*. Unfortunately, that term already has another meaning in our program specializer, namely the construction of a symbolic value given a value specification and a code expression, so we can't use it here.

```
(lambda (a b c d)
  (list 44 3 c (if b c d)))
```

and a return value template $(t_1\ t_2\ c\ t_3)$, where the value specifications of the symbolic values t_1, t_2, c, and t_3 are `44`, `3`, `(foo 1)`, and `(foo` $\top_{Number}$`)`, respectively. Additionally, the symbolic value t_3 has parents $\{c, d\}$. Thus, the return value specification for this call site, computed by taking the value specification attribute of all component symbolic values, is `(44 3 (foo 1) (foo` $\top_{Number}$`))`.

Assume that the specializer re-uses the specialization at another call site where `a` and `b` are unchanged, but `c=(foo 3)` and `d=(bar 3)`. We will use the names a', b', c' and d' to denote the symbolic values containing these value specifications. The substitution environment will bind the symbolic values in the index (*e.g.,* the formal parameters) to their new values (*i.e.,* $\{a = a', b = b', c = c', d = d'\}$). The return template is filled in follows. The first two elements, t_1 and t_2, are not bound in the substitution environment, and are copied unchanged. The third element, c, is bound to c', so that value is used. The fourth element, t_3, is not bound in the substitution environment, but was built via generalization. Looking up both of its `parents`, c and d, in the substitution environment yields c' and d' respectively; generalizing their value specifications gives a new symbolic value t_4 with value specification `(`$\top_{Symbol}$ `3)`. Thus, the "filled in" template is $(t_1\ t_2\ c'\ t_4)$, which has a value specification of `(44 3 (foo 3) (`$\top_{Symbol}$ `3))`, which is correct for the new parameter values.

This method allows the specializer to get as much information out of a call to re-used specialized procedure as it would have had it built a new specialization. Thus, it has the potential to reduce specialization time without compromising the accuracy of specialization. It is a type 3 modification; adding templates affects only the type inference portion of the specializer, not the re-use mechanism.

6.4.2 Handling Higher-Order Programs

All of the examples thus far have been limited to first order programs. In this section, we show that our mechanism can be made to work for higher-order programs, and describe the necessary changes to various higher-order specialization algorithms.

Specializers represent families of runtime closures by single closures at specialization time. These closure approximations contain an expression and an environment binding free variables to specialization-time approximations of runtime values. Closures are first-class objects and may be passed anywhere; if two closures of the same `lambda`-expression are

generalized, corresponding slots in the environment are generalized also. Handling higher-order programs requires that we produce accurate use information for values captured in closures, as well as values passed as arguments to closures. Since building a closure is a "non-touching" operation, it does not affect the MGIs of any values in "closed-over" variables; all of the MGI-lowering work happens when closures are unfolded or specialized. We treat unfolding first, then specialization.

Unfolding Closures

Unfolding a closure is similar to unfolding a first order procedure, in that the domain specifications of the actual parameters may be altered if they are used in the body of the closure. In addition, the domain specifications of values in the closure's environment (*i.e.,* values bound to its free variables) may also be altered. Both cases are treated identically; that is, whenever a value is used to perform a reduction, its domain specification is altered. Because unfolding the closure makes use of the closure's body and environment, the domain specification of the closure is lowered accordingly.

An additional update is necessary because, when we unfold a closure's body, we are making use of the fact that we know the closure's body; *i.e.,* the residual expression is valid only for other closures constructed from the same `lambda` expression. We note this by updating the domain specification of the symbolic value containing the closure to the closure's value specification.[10]

For example, consider an implementation of pairs using closures (Figure 6.4). If we specialize

```
(define (test x y)
  (+ (kar x) y))
```

on `x` bound to the result of `(kons 1 2)` and `y` unknown, and unfold both the application of the procedure `kar` and the closure bound to `x`, the residual code will be

[10]Of course, the application may or may not make use of particular values in the closure's environment. Setting the domain specification to the closure object (element of Val') says nothing about the use of values in the closure's environment. Instead, subsequent updating of the domain specifications of symbolic values bound in the environment will automatically be reflected whenever we examine the head's domain specification, since it *contains* the relevant symbolic values, which are dereferenced through their domain specifications when we compute the MGI. This is an example of why domain specifications are elements of Val' rather than elements of Val.

```
(define (kons the-kar the-kdr)
  (lambda (msg)
    (case msg
      ((kar) the-kar)
      ((kdr) the-cdr))))

(define (kar the-kons)
  (the-kons 'kar))

(define (kdr the-kons)
  (the-kons 'kdr))
```

Figure 6.4: An implementation of pairs using closures and message passing

---

```
(define (res-test x y)
  (+ 1 y))
```

where the domain specification for `x` is a closure of `(lambda (msg) ...)`, where the free variables `the-car` and `the-cdr` have domain specifications `1` and $\top_{Val}$, respectively. The variable `y` has domain specification $\top_{Val}$. Thus, the specialization `res-test` could be re-used for `x` bound to the result of `(kons 1 3)`.

Thus, we see that values in the environment of a closure which is unfolded are treated analogously to values contained in a data structure which is accessed, while values passed as actual parameters in an unfolded call to a closure are treated like actual parameters to a first-order procedure which is unfolded. This requires no modifications to the re-use criterion or to the specializer, except for updating the domain specification of the call head when it is known to denote a particular closure (a type 2 modification).

Specializing Closures

The previous subsection described how unfolded closures (which are completely consumed at specialization time) are treated, leaving open the question of how specialized closures (which appear in the residual program) are handled under our re-use mechanism. Specifically, we must decide when and how the domain specifications of actual parameter values in residual applications and free variables of residual closures should be modified.

The proper choices depend on how specializations of closures are constructed. Most existing specializers for higher-order Scheme [115, 23, 45, 11] build residual closures by

specializing their bodies on completely unknown argument vectors. No knowledge of the argument values at the residual call sites of a closure is used in computing its specialization; thus, there is no need to lower the domain specifications of the actual parameters at any of the closure's residual call sites.

However, the value domain specifications of the free variables may be altered during the specialization of the closure. This (correctly) restricts the applicability of the enclosing specialization because (due to arity raising) all unknown symbolic values in the closure's environment refer to formal parameters of the enclosing specialization. The only problem is one of timing; the MGI of the enclosing specialization (which binds the closure's free variables) will not be valid until the specialization of the closure is completed. This poses no difficulties for FUSE, which specializes a closure as soon as it finds it passed into a potentially residual context (*c.f.* Section 3.3.5); this determination always takes place during the construction of the enclosing specialization. Solutions based on traversing the residual program to find and specialize closures in residual contexts (such as that suggested by Mogensen in [83]) may encounter problems, since specializing a closure could "lower" the MGI of the enclosing specialization in a way that invalidates some applications in the (already constructed) residual program.

The version of FUSE described in Chapter 5 of this dissertation, Section 4 of [97] and in [94] uses control flow analysis to compute more accurate argument vectors for specialized closures. We have not implemented specialization re-use under this algorithm, but believe that doing so would not be difficult; the remainder of this subsection describes how this could be done.

The control flow analysis approach still builds only one specialization per closure, but builds a more accurate specialization. The argument values at residual call sites of a specialized closure are indeed used in constructing the body of the specialized closure, but they do not affect the construction of the residual call, or any other code in the specialization containing the call. Thus, there is still no need to lower the domain specifications of the arguments at residual call sites. This differs from the construction of residual calls to specializations of first-order procedures, where we must lower the domain specifications of the actual parameter values to match those of the index of the specialization. The reason for the difference is that, in the first order case, the specializer uses the actual parameters to choose one of multiple specializations of the called procedure; different argument values could thus result in a different residual call. In the higher-order case, there is no specialization-time

choice because there is still only one specialization per closure, and the determination of which specialization to invoke at a call site is not made until runtime.

However, even though the re-use mechanism doesn't have to be changed, the iterative algorithm for computing the bodies of specialized closures does require a slight alteration to work properly in the presence of the re-use mechanism. The algorithm operates by incrementally determining, during the specialization process, which residual call sites are reached by each closure. Each time a closure reaches a new call site, the argument specification of the specialized closure is recomputed by generalizing it with the argument specification of the call site; if the specialization's argument specification changes, the specialization is rebuilt. The correctness of this algorithm relies on the fact that the argument specification (*i.e.,* the vector obtained by taking the value specifications of each argument) for each call site specifies all values which could be passed to the specialized closure from that site at runtime, so that generalizing the specifications from all call sites will yield a specialization index which is sufficiently general. The re-use mechanism invalidates this assumption, because if a call site's enclosing specialization is re-used, different values (not reflected in the call site's argument specification) may reach the call site, but will not be seen by the iterative algorithm, resulting in an insufficiently general specialization of the closure.

Either of two possible modifications will solve this problem. First, the domain specifications of the arguments at a call site do correctly (though not necessarily precisely) approximate all values which could be passed at that site at runtime. Thus, we can simply modify the iterative algorithm to use domain specifications (once they are completely "lowered," which is true after the call site's enclosing specialization is complete) instead of argument specifications. Because the domain specifications at a call site approximate all values which might appear at that site as a result of any possible reuse of the call site's enclosing specialization, this approach can risk building an overly general specialization of a closure. The problem is that the domain specifications must represent all values which might arrive due to any legal re-use, not only those values which will actually arrive at runtime. For example, specializing

```
(define (g f x)
  (if (number? x)
      (f x)
      x))
```

on `f=`$\top_{Val}$ and `x=4` will compute a domain specification of $\top_{Number}$ for `x`, since the specialization can safely be re-used for any numeric `x`. Thus, the argument vector computed for any closure that might reach the call site `(f x)` will be at least as high as $\top_{Number}$. If the specialization of `g` is re-used on `x=3` and `x=5.5`, this will be optimal; however, if `g` is re-used only on integers, the call site `(f x)` should really only constrain the specializations of closures reaching the site to be specialized on $\top_{Integer}$.

We can solve this problem by not using the domain specifications when computing the argument specifications for specialized closures. Instead, each time a specialization is re-used, we must recompute the value specifications of all residual higher-order call sites within the body of the specialization, and use these specifications to recompute the argument specifications of any closures reaching the sites. This can be accomplished either by a template mechanism similar to that used for computing return values in Section 6.4.1, or by extending the incremental control flow analysis mechanism of [97, Section 4.5] to pass information about scalar values, in addition to pairs and closures, along initial source/final destination links. In any case, the modifications necessary to make the re-use mechanism work with higher-order specialization affect only the higher-order specialization mechanism, not the re-use mechanism (type 3 modification).

6.4.3 Improving Re-use

In the previous subsections, we have described how to extend the re-use mechanism to deal with type inference and higher-order extensions to the specializer. We now turn to the problem of improving the re-use mechanism itself, either to reduce the number of redundant specializations constructed and discarded, or to increase the number of opportunities for sharing through re-use.

Improving the Re-Use Criterion

As described in Section 6.3.3, our re-use criterion will re-use a prior specialization without first building a new one when it can prove that (1) it is safe to do so, and (2) building a new specialization would not be worthwhile. (1) is easily proved or disproved using the safety criterion of Definition 3, but (2) can be proved in only limited cases. The algorithm of Section 6.3.3 can prove (2) only when the new argument specification v' is more general than the old one v (*i.e.*, $v \sqsubseteq v'$). This is possible because more general values *always* lead to more general MGIs.

Unfortunately, this criterion is rather weak, and can cause the specializer to waste time building and discarding specializations. Two cases in which this occurs are as follows. First, a more specific argument value may not allow the specializer to build a better specialization: *i.e.,* specializing

```
(lambda (x) (number? x))
```

on `1` yields the same MGI as specializing it on $\top_{Number}$. If we build the specialization for `1` first, all is well, but if we build the one for $\top_{Number}$ first, our algorithm has no way of knowing that `1` won't lead to any more reductions; it must evaluate `(number? 1)` in order to determine this, at which point it finds out it needn't have bothered. This is exactly the problem we saw in the second example of Section 6.3.3 (page 173). We call this the problem of *refined indices.*

A similar problem arises even when the new argument value is not more specific than the first one, but merely incomparable to it. Specializing the function above on `1` gives an MGI of $\top_{Number}$; on a subsequent attempt to specialize it on `2`, the algorithm cannot determine that `number?` will treat `1` similarly to `2` without actually evaluating it. This can be a real problem in interpreters which pass explicitly tagged data structures, some of whose tags are never examined; since all of the tags are known symbols (incomparable in our type system), the specializer must build and discard specializations for each combination of tags in unexamined positions. As a trivial example, consider

```
(lambda (obj)
  (if (eq? (first obj) 'pair-tag)
      (second obj)
      'error))
```

specialized first on `obj=(pair-tag (num-tag` $\top_{Val}$`) (num-tag` $\top_{Val}$`)))` and then on `obj=(pair-tag (sym-tag` $\top_{Val}$`) (num-tag` $\top_{Val}$`))`. Both specializations use only the information that the argument, `obj`, is a list with at least two elements, and whose first element is the symbol `pair-tag`; thus, the MGI is `(pair-tag` $\top_{Val}$ `.` $\top_{Val}$`)`. Unfortunately, because `sym-tag` and `num-tag` are incomparable, the $v \sqsubseteq v'$ test fails, and we must build the second specialization, only to find that it has the same MGI, and discard it. We will call this the problem of *incomparable indices.*

In both cases, our algorithm saves no work (*i.e.,* fails to prove that the new specialization will have the same MGI as the old one) because the MGI does not denote just the set of

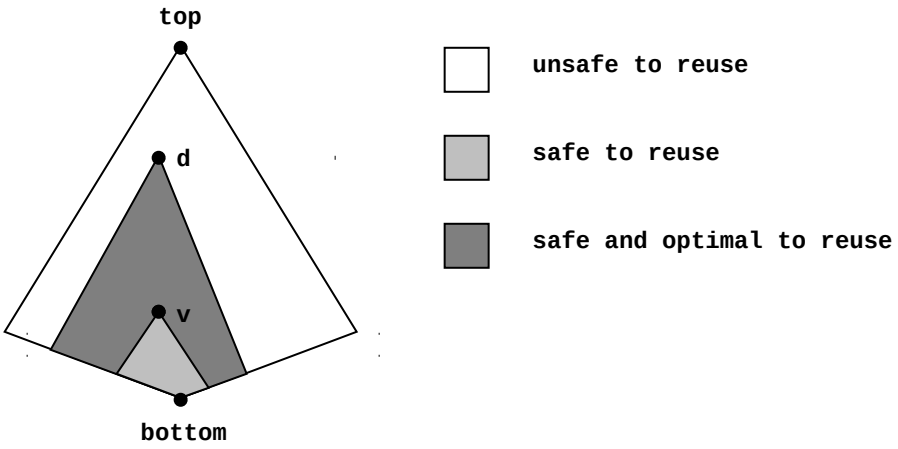


Figure 6.5: Re-use criterion, extended to take advantage of the properties of incomparable indices. Given an argument specification v', the reusability of an existing specialization with argument specification v can be determined by locating v' in the diagram above. Re-use is safe when $v' \sqsubseteq d$ and is optimal when $v' \not\sqsubseteq v$.

---

values for it is *optimal* to re-use the specialization, but rather the entire set of values for which such use is *safe*. In this section, we will describe two ways of addressing these cases. Our first solution limits the behavior of primitive reductions in the specializer in order to solve the problem of incomparable indices, but doesn't address the problem of refined indices. Our second solution applies to both cases without limiting the behavior of the specializer, but requires that extra information be maintained.

The problem with incomparable indices arises when the specializer constructs a specialization on a specification v, resulting in MGI d, and is subsequently asked to compute a specialization on a specification $v' \sqsubseteq d$ where $v \not\sqsubseteq v'$ and $v' \not\sqsubseteq v$. If v' denoted strictly less information than v, we would be able to establish that the new specialization could be no better than the prior one (*i.e.*, if we denote the new specialization's MGI by d', $d' = d$), while if v' denoted more information, the new specialization might be better (*i.e.*, $d' \sqsubseteq d$). Unfortunately, neither of these relationships holds here. The re-use algorithm cannot make any determination, and so must build a new specialization, then compare the two MGIs.

We solve this dilemma by requiring that specializations built on incomparable values

have MGIs which are either equal or incomparable; we call this the *incomparability restriction*. Adopting this restriction makes the optimal re-use test decidable for all incomparable values. Consider building a new specialization on the new value v'. If $v' \not\sqsubseteq d$, re-use will be unsafe, and a new specialization must be constructed. If $v' \sqsubseteq d$, then, by the properties of specialization, we know that $d' \sqsubseteq d$ (where d' is the MGI of the new specialization on v'). Adding the incomparability restriction tells us $d' = d \vee (d' \not\sqsubseteq d \wedge d \not\sqsubseteq d')$; combining these properties gives $d' = d$, so we can conclude that the new specialization will be redundant. Thus, for incomparable values v and v', re-use is always either unsafe or optimal; we will never have to build a specialization for v' and discard it.

This allows us to change the re-use criterion to re-use the existing specialization whenever $v' \sqsubseteq d \wedge v' \not\sqsubset v$ (for a pictorial representation of this criterion, see Figure 6.5).[11] For example, a specialization of

```
(lambda (x) (number? x))
```

built on `x=1` (with MGI $\top_{Number}$) could be re-used for `x=2`, because $2 \sqsubseteq \top_{Num} \wedge 2 \not\sqsubset 1$. Similarly, applications of

```
(lambda (obj)
  (if (eq? (first obj) 'pair-tag)
      (second obj)
      'error))
```

with different values for `(second obj)` would share the same specialization. This criterion is not useful for refined indices; *e.g.,* a specialization of

```
(lambda (x) (number? x))
```

built on `x=`$\top_{Number}$ could not be preemptively re-used for `x=1`, because $1 \sqsubset \top_{Num}$.

The question at hand is whether the incomparability restriction constrains the specializer too strongly. Basically, it forces applications of primitive operators in the specializer to treat siblings in the domain of values consistently: it must never be the case that one subtype of a type is used completely, while another is not used at all. For example, in the numeric subdomain of Figure 6.6, any primitive which uses an argument of $\top_{Integer}$ (*i.e.,* lowers

[11] This is the solution originally used in FUSE, and documented in the original version of [92]. The revised version of that paper [91] switched to the criterion of Section 6.3.3, which is applicable to all specializers, not just FUSE.

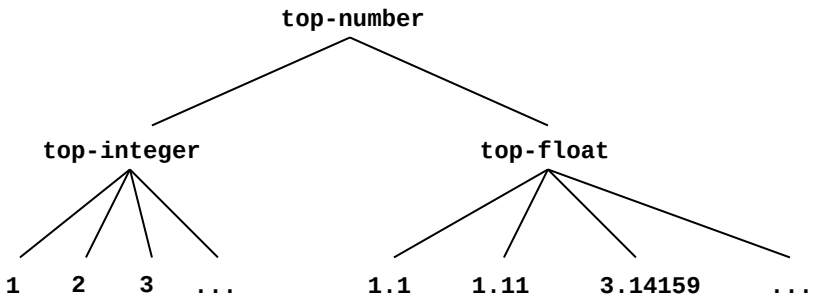


Figure 6.6: A subdomain for numeric types

its domain specification to $\top_{Integer}$) would also have to use an argument of $\top_{Float}$. An argument of `5.5` wouldn't have to be completely used, but its domain specification would have to be lowered to at least $\top_{Float}$. For example, the predicate `integer?` has the desired behavior (it uses the fact that its input either is or is not an integer). Conversely, any primitive which doesn't make use of all of the information in a particular value must not make use of all of the information in any incomparable argument value (*e.g.,* since `number?` uses only $\top_{Number}$ when applied to `5`, it must also use $\top_{Number}$ when applied to `6`, $\top_{Float}$, or `5.5`).

This property holds for most reasonable implementations of specialization-time primitives. We have encountered only one optimization which violates this property: algebraic simplification. Consider an implementation of `+` which reduces to one of its arguments when the other argument is is known to be `0`. Applying `+` to `1` and $\top_{Integer}$ yields an MGI of $(\top_{Integer}\ \top_{Integer})$[12], while applying it to `0` and $\top_{Integer}$ yields an MGI of `(0` $\top_{Integer}$`)`. This is not a problem in the current incarnation of FUSE, which does not perform such simplifications; specializers with such optimizations could not use this approach.[13]

[12] This is assuming that we are limiting specialization by forbidding the inlining of the constant `1` in this case; for details, see the discussion of limiting in Section 6.4.3.

[13] Actually, this criterion works even in the presence of algebraic simplifications, provided that a sufficiently detailed type lattice is used. Consider a case where, in addition to lattice elements for $\top_{Integer}$ and the concrete integers 1, 2, etc., the lattice contained an element $\top_{Nonzero\text{-}integer}$. Then applying `+` to `1`

The incompatibility restriction is both a type 1 modification, in that it modifies the re-use criterion, and a restriction on the operation of the specializer (*i.e.*, it cannot violate the incompatibility restriction), which might require a type 3 modification to specializers which don't already have the incompatibility restriction property.

A second approach, which addresses the problem of refined values in addition to that of incomparable values, uses a different tactic. Instead of constraining the specializer (via the incomparability restriction) so that re-use decisions can be made using the argument values and the MGI alone, this approach computes additional information which allows the specializer to prove that re-use of a specialization is optimal. Remember that we want to know whether, given a specialization constructed on argument specification v, yielding MGI d, building a new specialization on v' where $v' \sqsubseteq d$ might be worthwhile. It isn't worthwhile for $v \sqsubseteq v' \sqsubseteq d$ (*c.f.* Section 6.3.3), but we need a solution for the cases $v' \sqsubseteq v$ and $v \not\sqsubseteq v'$. We accomplish this by keeping track of not only the information used in constructing a specialization, but also what (unavailable) information would allow a better specialization to be constructed. For example, specializing

```
(lambda (x) (number? x))
```

on $\top_{Number}$ yields an MGI of $\top_{Number}$, and notes that no amount of additional information would be useful (in building a new specialization). Thus, building a new specialization for `1` would be a waste of time. Similarly, specializing

```
(lambda (x) (+ x 5))
```

on $\top_{Integer}$ yields an MGI of $\top_{Integer}$, and notes that any value v for `x` where $v \sqsubset \top_{Int}$ would lead to a better specialization. Thus, new specializations would be built for `x=1`, `x=2`, etc.

There are several possibilities for representing this notion of useful additional information. One might consider computing a set of all values which would be useful; this is unrealistic because this set would often be infinite (in the previous example, all integers are useful). A more practical method approximates the set of useful values by its least upper bound in the lattice; the idea is that any value lying at or below this bound is potentially useful, while any value lying above this bound is guaranteed to yield the same MGI as that

---

and $\top_{Integer}$ would give an MGI of $(\top_{Nonzero\text{-}Integer}\ \top_{Integer})$, while applying `+` to `0` and $\top_{Integer}$ would yield an MGI of $(0\ \top_{Integer})$. Because $\top_{Nonzero\text{-}Integer}$ and `0` are incomparable, the MGIs are incomparable, and the incomparability restriction holds.

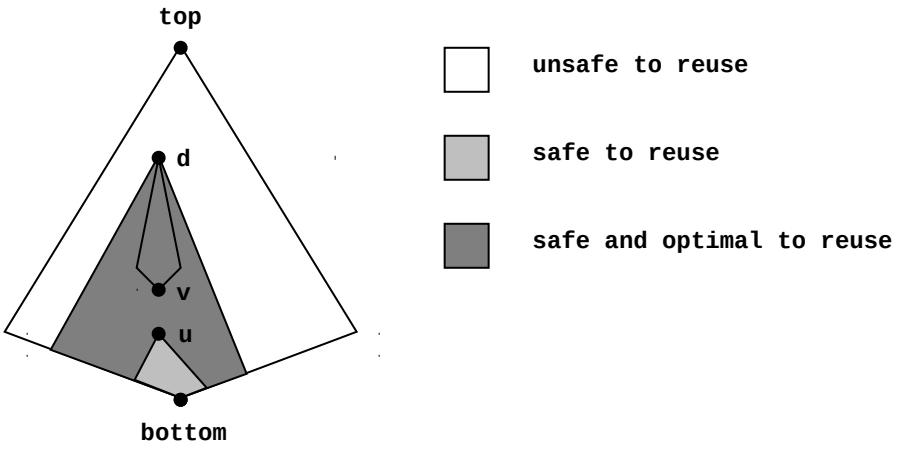


Figure 6.7: Re-use criterion, extended with an approximation of "useful" information. Given an argument specification v', the reusability of an existing specialization v can be determined by locating v' in the diagram above. Re-use is safe when $v' \sqsubset d$ and is optimal when $v \sqsubseteq v'$ or $v' \not\sqsubseteq u$. The graphic above depicts a case where $u \sqsubset v$; this is not necessarily the case.

---

of the specialization associated with the bound.[14] We will call this bound the "demand specification," as it represents information that is demanded but unavailable. Given a specialization built on value v with MGI d and demand specification u, we can avoid building a new specialization for another argument value whenever re-use is safe ($v' \sqsubseteq d$), and either

- The new value is more general than the old one ($v' \sqsupseteq v$), or
- The new value is not more specific than the most general "useful" value ($v' \not\sqsubseteq u$).

This criterion is depicted pictorially in Figure 6.7. For example, specializing

```
(lambda (x) (number? x))
```

on either $\top_{Number}$ or `1` would yield an MGI of $\top_{Number}$ and a demand specification of `bottom`, indicating that all non-bottom values which are numbers will result in the same

[14] This is still somewhat coarse, as there is no way to say that all known integers are useful without also saying than $\top_{Integer}$ is useful also (though, if, as in the example, $\top_{Integer}$ is the MGI, we know it isn't useful). Maintaining such a distinction would require the use of a domain that models sets of values taken from the specializer's value domain, including partially known elements like $\top_{Integer}$; *e.g.,* the powerdomain of *Val* rather than *Val*.

specialization. Thus, unlike the criterion of Figure 6.5, we obtain the desired result (re-use of the specialization) for both refined indices (first specializing on $\top_{Number}$, then on `2`), and for incomparable indices (first specializing on `1`, then on `2`). Furthermore, this criterion operates correctly under specializers that perform algebraic optimizations. Specializing

```
(lambda (x y) (+ x y))
```

on `x=1` and `y=`$\top_{Integer}$ yields the specialization

```
(lambda (x y) (int-+ x y))
```

which has a MGI of (`⊤`$_{Integer}$ `⊤`$_{Integer}$) and a demand specification of (`⊤`$_{Integer}$ `⊤`$_{Integer}$). Thus, the specialization can be re-used for `x=`$\top_{Integer}$, `y=`$\top_{Integer}$ (because only the fact that both `x` and `y` were integers was used in building the specialization), but not for `x=0`, `y=`$\top_{Integer}$, because that argument vector would yield a different specialization, namely

```
(lambda (x y) y).
```

We can simplify our implementation somewhat by noting that, in practice, it is almost always the case that the demand approximation is either equal to the MGI (meaning that values below the MGI would be useful) or bottom (meaning that all values below the MGI are useless, except for bottom, which is always useful for strict primitives). Cases where this is not true do exist; for example, siblings in the type lattice may not be equivalently useful in performing reductions. Consider a lattice which contains $\top_{Number}$, $\top_{Fixnum}$, and $\top_{Bignum}$ (where $\top_{Fixnum} \sqsubset \top_{Num}$ and $\top_{Bignum} \sqsubset \top_{Num}$) and and an implementation of `1+` which can make use of the fact that its argument is a fixnum, but not that it is a bignum. Applying such an operator to $\top_{Bignum}$ would return a domain specification of $\top_{Number}$ and a demand specification of $\top_{Fixnum}$. However, we have yet to encounter any such cases in our partial evaluators.

This allows the simplification of reducing the demand specification to a single boolean flag per symbolic value. If the flag is true, then the demand specification is the same as the domain specification (*i.e.,* values below the domain specification but above bottom are useful); otherwise, the demand specification is `bottom` (the only useful value below the domain specification is the bottom element). Primitives which receive an argument which is sufficiently known to allow them to be reduced completely (*e.g.,* `number?` applied to $\top_{Number}$) set the flag to false; otherwise, the flag is set to true. The re-use criterion under

this simplification is as follows. Given a specialization built on value v with MGI d, and with demand function f mapping a symbolic value to its demand flag, we can avoid building a new specialization for another argument value whenever re-use is safe ($v' \sqsubseteq d$), and either

- The new value is more general than the old one ($v' \sqsupseteq v$), or
- The specialization is optimal for all non-bottom values, and the new value is not bottom (for all subparts x of v', $\neg f(x) \wedge (x \neq \bot)$).

This criterion allows us to handle the second example of Section 6.3.3 without having to build and then discard a specialization built on `n=1`.

Computing "useful" information is both a type 1 and a type 2 modification; not only do we change the re-use criterion, but we must compute new information (demand specifications) to evaluate the criterion.

Limiting Specialization

In the previous subsection, we increased the amount of re-use by improving the accuracy of the re-use criterion, allowing it to avoid constructing new specializations in a greater number of cases. In this subsection, we describe another way to achieve more re-use, not by improving the estimation of the specializer's use of information, but by limiting that use; *i.e.,* constructing more general specializations.

The re-use analysis described above operates by determining which of the information present in the original argument specification (specialization index) is used, and re-using the specialization on argument specifications for which it can prove that the same information will be used. What should "used" mean? We have been using the criterion that information about a value is used when it allows the specializer to perform a reduction at specialization time that otherwise would have to be performed at runtime. Using information in a value specification to decide a conditional, apply a function call head, or execute a primitive procedure are all clearly instances of this criterion. What is more difficult to decide is what to do when given information about a primitive procedure's arguments, but not enough to execute the primitive. Such an example is specializing

```
(lambda (x y)
  (+ x y))
```

with `x` specified as `2` and `y` specified as $\top_{Val}$. Our choice is between producing the specialization `(lambda (x y) (+ 2 y))`, restricting the domain to `x` equal to `2`, or producing `(lambda (x y) (+ x y))`, which doesn't restrict its domain. Both specializations return the same information (namely, that the result of applying it is a number), and neither one completely reduces the application of `+`. The question is: "Is the first specialization better?" In our view, this depends on the virtual machine that will be executing the residual program. Some processors have an "add immediate" instruction, which can add the constant 2 to a register value faster than it can add two register values, some take the same amount of time for both operations, and some might even run more slowly on the first specialization because of the overhead of loading the value `2` into a register. Also, building residual code of the form `(+ 2 x)` makes fewer specializations re-usable, increasing the size (and slowing the performance) of residual programs. One way out of the problem would be to delegate it to the code generation postpass, which could be expected to know more about the target architecture; it could detect and merge specializations which it considered to be the same. However, this would remove an opportunity for speeding up the specializer; since, if the specializer knew these rules, it could avoid building redundant specializations such as `(lambda (x y) (+ 2 y))` and `(lambda (x y) (+ 3 y))`, instead of building both and having the code generator "clean up" later.

Another opportunity for sharing comes from *non-touching* constructor primitives such as `cons`, which can be reduced[15] at specialization time without any knowledge about its arguments. At specialization time, the reduction of `(cons x y)`, where `y` is `1`, doesn't "use" the information that `y` is `1`; the question is whether `(cons x 1)` is better code than `(cons x y)`, where `y` will be bound to `1` at runtime. This situation often occurs when the usual Scheme `append` routine is specialized: each call site with an unknown first argument and a known second argument will generate a different specialization, yet all of these specializations will be identical save for a single constant in a residual application of `cons`. Is the benefit of having constants inlined worth the cost of a much larger residual program?

We are uncommitted on this issue, and believe that further research is necessary to determine when the benefits of performing a particular specialization outweigh its costs. In order to facilitate such work, FUSE is configurable: the behavior of each FUSE primitive, when faced with inputs on which it cannot be reduced at specialization time, can be individually tuned. Configuring primitives to use their arguments (*i.e.* alter their arguments'

[15] Technically, both reduced and left residual; see [114] for an explanation.

domain specifications) only when necessary to perform a reduction leads to more sharing, while configuring them to take advantage argument information for code generation leads to less sharing, but possibly faster residual programs.

In any case, limiting specialization affects only the specializer's reduce/residualize choice (*i.e.,* type 3 modification), and does not require that we alter either the re-use criterion or the computation of demand specifications.

6.5 Discussion

We have devoted much space to formalizing the problem of redundant specialization, and to describing a solution (and its variants) to the problem. However, the examples in the introduction are all very small; this begs the question of whether redundant specialization is indeed a problem in practice. We now address this issue by showing several examples where it is indeed a problem, and argue that, as specializers become more powerful, it will become more severe.

6.5.1 Examples

In this section, we describe two examples of redundant specialization which occur when specializing realistic programs with FUSE. Our first example, an interpreter for an imperative language, shows an instance of a problem which affect all specializers, while our second example shows how our re-use mechanism removes some forms of overhead which are specific to online specialization techniques.

Interpreters

Our first example is an interpreter for a small imperative language. The "MP" language, first used as an example by Sestoft [102], has `if`, `while`, and assignment statements, as well as a variety of expressions. A fragment of a direct-style interpreter for this language is given in Figure 6.8. The MP program shown in Figure 6.9 computes the x^y where x and y are unary numbers represented as lists of arbitrary tokens.

We would expect that specializing an MP interpreter on this program would build two residual loops (*i.e.,* specializations of the procedure `mp-while`), one for both of the two `(while kn ...)` loops, and one for the `(while next ...)` loop. Most offline specializers (such as those described in [12, 76, 83, 102] do exactly that. They are able to build one

```
(define mp
  (letrec
     ((init-store ...)
      (mp-command
       (lambda (com store)
         (let ((token (car com)) (rest (cdr com)))
           (cond
            ((eq? token ':=)
             (let ((new-value (mp-exp (cadr rest) store)))
             (update store (car rest) new-value)))
            ((eq? token 'if)
             (if (not (null? (mp-exp (car rest) store)))
                 (mp-command (cadr rest) store)
                 (mp-command (caddr rest) store)))
            ((eq? token 'while) (mp-while com store))
            (else ;(eq? token 'begin)
             (mp-begin rest store))))))
      (mp-begin
       (lambda (coms store)
         (if (null? coms)
             store
             (mp-begin (cdr coms) (mp-command (car coms) store)))))
      (mp-while
       (lambda (com store)
         (if (mp-exp (cadr com) store)
             (mp-while com (mp-command (caddr com) store))
             store)))
      (mp-exp ...)
      (lookup-proc ...)
      (update ...)
      (lookup ...)
      (main ...)
     main))
```

Figure 6.8: Fragment of an interpreter for MP

```
(program (pars x y) (dec out next kn)
  (begin
    (:= kn y)
    (while kn
           (begin
             (:= next (cons x next))
             (:= kn (cdr kn))))
    (:= out (cons next out))
    (while next
           (if (cdr (car next))
               (begin
                 (:= next (cons (cdr (car next)) (cdr next)))
                 (while kn
                        (begin
                          (:= next (cons x next))
                          (:= kn (cdr kn))))
                 (:= out (cons next out)))
               (begin (:= next (cdr next))
                      (:= kn (cons '1 kn)))))))
```

Figure 6.9: Exponentiation program written in MP

residual loop implementing the two `(while kn ...)` loops because the interpreter procedure implementing `while` loops (`mp-while`) is invoked on identical arguments for both loops; *i.e.,* the expressions for the loop bodies are the same, the identifiers in the store are the same, and the values in the store have all been abstracted to $\top_{Val}$ ("dynamic," in offline terminology).

Online specializers such as `FUSE`, however, do not obtain this result. Because online specializers perform abstraction only when necessary for termination purposes, they compute approximations to the values in the store. In particular, FUSE is able to deduce that the value bound to `out` is the empty list when the first `(while kn ...)` loop runs, but is a pair when the second `(while kn ...)` loop runs. Thus, the store argument to `mp-while` is different at the two call sites, causing the specializer to build two distinct specializations, one for a store with `out` known to be the empty list, and one for a store with `out` known to be a pair. Of course, since the `(while kn ...)` loops never assign or reference the variable `out`, the two specializations are identical. This wastes time in the specializer, which must construct an extra specialization, and produces a residual program containing two identical specializations.

When we enable the re-use mechanism, things are different. After the specializer builds the first specialization for `(while kn ...)` where `out` is known to be the empty list, the domain specification of the corresponding cell in the store is $\top_{Val}$, indicating that the information in the call, `nil`, was unused. When asked to build a specialization where `out` is known to be pair, the specializer can deduce that it is safe to re-use the existing specialization (because $(\top_{Val} \ . \ \top_{Val}) \sqsubseteq \top_{Val}$). If either of the optimality criteria of Section 6.4.3 are used, specializer will immediately deduce that the new value for `out` will not be useful, and will re-use the existing specialization, while if the simpler criterion of Section 6.3.3 is used, it will build a new specialization, discover that it has the same MGI as the old one, and then discard it.

This sort of difficulty is common in interpreters that maintain a store or environment representing the state of the program being interpreted. Loops in the interpreted program will force the construction of residual versions of interpreter procedures which take this state as an argument. Since most loops in a program reference only a small fraction of the variables in the state, we would expect to see many redundant specializations based on the values of unreferenced variables in the state.

However, the literature on specializing interpreters [12, 22, 33, 102, 64] fails to report such redundancies. We believe there are two reasons for this. First, most work to date on specializing interpreters has been performed with offline specializers. Such specializers treat all values which are not guaranteed to be known at specialization time as $\top_{Val}$; thus, the values in the state are represented either by a single $\top_{Val}$ value (in the case of non-partially-static binding time analyses), or a $\top_{Val}$ value per element of the state (in the case of partially-static BTA). In either case, the values (as opposed to the names) bound in the state are unavailable at specialization time, and cannot lead to redundant specializations. Second, most of the interpreters specialized to date have implemented small imperative languages without procedure call. When specializing an interpreter for a language without procedure call, there are few opportunities for sharing. That is, two loops in the interpreted program can be implemented by a single residual loop only when their bodies are textually identical (since their text is used in building the residual loop), which is rare. Given a language with procedures, a loop can be abstracted into a procedure and executed under multiple contexts at specialization time, leading to redundant specialization if these contexts lead to identical reductions in the body of the loop. For example, in a language with procedures, the two `(while kn ...)` loops in the program of Figure 6.9 would most likely

	no reuse	reuse	improvement
specialization time (msec)	4867	4816	1.0%
code generation time (msec)	310	299	3.5%
residual program size (pairs)	1180	1024	13%
# procedures constructed	5	3	40%
# procedures in residual	4	3	25%

Figure 6.10: Results of specializing MP interpreter on exponentiation program, with and without re-use mechanism. Times are process times (not including gc time) for MIT CScheme on a 28MB NeXT workstation. The "no reuse" column is for a specializer which does not compute domain specifications at all; computing domain specifications (but not using them) takes 5658 msec.

---

be abstracted into a single procedure, which would then be invoked on two different (but, for purposes of specialization, identical) stores.

One might also wonder if the need for re-use technology when specializing interpreters is limited to online specializers, since, in our example, an offline specializer's binding time analysis would simply have abstracted away the values in the state. This is certainly not the case; it is easy to envision interpreters for which offline specializers also build redundant specializations. Consider a statically typed language with parametric polymorphism. An interpreter for such a language would either factor the type descriptors into a separate type environment which is completely known at specialization time (as is done by the Algol interpreter in [27]), or would leave them attached to the values in the state (which would still allow a sufficiently precise Binding Time Analysis to deduce that they are static). In either case, the known values used by the interpreter to implement types would not be abstracted by the binding time analysis and could thus lead to redundant specialization. Offline program specializers for typed languages can treat polymorphism in programs explicitly [76], but even such specializers would need a re-use mechanism for interpreters for polymorphic languages, due to the need to represent values in the interpreted program using a single universal type.

The results of specializing the interpreter of Figure 6.8 on the program of Figure 6.9 with and without the re-use mechanism are shown in Figure 6.10. Re-using the same specialization of `mp-while` for both `(while kn ...)` loops yields a 13% smaller residual program with 1 less specialization. The time savings are less impressive. The specialization criterion

avoided building two specializations (one is the redundant `while` loop described above; the other is an artifact of the specialization technique, and is explained in Section 6.5.1), but saved only 1% in specialization time. This is due to overhead in computing domain specifications, performing extra type comparisons, and instantiating return value templates. In this example, where the specializer spent approximately 16% of its time on such overhead, the time saved by not building the two avoided specializations is approximately equal to the overhead, yielding a better-quality residual program but no time savings. We expect to see greater time savings in the future, for several reasons. First, the redundant specialization in this example was quite small; larger programs may have more, larger, redundant specializations, giving greater savings. Second, the amount of overhead is heavily dependent on the complexity of the specializer; as we move to more complex termination mechanisms, etc, the amount of time spent computing domain specifications will become a smaller fraction of the total specialization time. Finally, the type comparison and return value instantiation operations in our prototype implementation are very slow; we expect that clever programming could speed them up by almost an order of magnitude.

Online Specialization

In the interpreter case above, the risk of redundant specialization arises from the nature of the program being specialized; the interpreter contains a procedure (`mp-while`) which, due to the structure of the MP program, is invoked on different but (for purposes of specialization) equivalent arguments at specialization time.

Sometimes, the risk of redundant specialization comes from the structure of the specializer, which specializes a particular procedure multiple times even though it is only invoked once at specialization time, and only one specialization of the procedure will appear in the residual program. In FUSE, this arises because of the fundamentally iterative nature of the specializer.

First, FUSE uses pairwise generalization [112, 100, 96] to compute what actual arguments to use when building a specialization. That is, FUSE will compute the body of a specialization over and over again on increasingly general arguments until the arguments are sufficiently general that all non-unfoldable recursive calls to the same procedure within the body of the specialization match some specialization in the cache. Thus, expressions in the body of the procedure being iteratively specialized may be evaluated on more specific values during early iterations than during the final iteration. If those expressions include

procedure calls, each call site will see increasingly general arguments as iteration proceeds, meaning that any specializations built by a call site during an early iteration may be too specific for the same call site during a later iteration. Only the specializations built during the last iteration are guaranteed to appear in the residual program; the others may just be wasted effort.[16] For example, specializing

```
(define (foo x y)
  (if (= x 0)
      (bar y)
      (baz (foo (- x 1) (* x y)))))
```

on `x=`$\top_{Number}$ and `y=1` will first specialize the body of `foo` on `x=`$\top_{Number}$, `y=1`, then on `x=`$\top_{Number}$, `y=`$\top_{Number}$. Thus, we will construct two specializations of `bar`, one for `y=1` and one for `y=`$\top_{Number}$, even though only the latter specialization will be invoked by the specialization of `foo`.

Similarly, the analysis of Chapter 4 iteratively rebuilds the body of a specialization until its return value approximation converges. If the body of a procedure p invokes some other procedure q on p's return value, then q may end up being repetitively specialized on increasingly general values. Since the only specialization of q invoked by the specialization of p is the one built on p's final return value approximation, all but that final specialization may be useless. For example, if we assume that, in the example above, `bar` and `baz` are strict functions mapping numbers to numbers, then `baz` will be specialized first on `bottom`, then on `1`, and, finally, on $\top_{Number}$. Only this last specialization will be invoked by the residual version of `foo`.

Both of these situations result in the construction of potentially useless specializations. That is, if a specialization s is built on argument values v at a call site c, either (1) the arguments at c in the residual program are more general than v, rendering s inapplicable at c, or (2) c does not appear in the residual program at all. Of course, there may be some other call site c' with argument values v' in the residual program at which s is applicable, in which case effort was not wasted.

The re-use mechanism increases the chance that such a site c' will exist by allowing s to be re-used at sites where $v \sqsubseteq v' \sqsubseteq MGI[s]$[17] instead of only at sites where $v' = v$.

[16] We say "may" be wasted because some other portion of the program might indeed require a specialization constructed on one of the values that was too specific for this site, in which case the work is not wasted.

[17] Alternately, we could use either of the more sophisticated criteria of Section 6.4.3.

Indeed, s may be reusable at the same call site where it was constructed (*e.g.*, $c' = c$). For example, if `bar` and `baz` only use the fact that their argument is a number, then the (formerly "orphaned") specializations `(bar 1)` and `(baz 1)` can be invoked by the final specialization of `foo`. Even if this is not the case, the chance that the "orphaned" specialization will be usable somewhere else in the residual program is increased.

Such opportunities for re-use do occur in realistic programs. On such example is the second redundant specialization detected when specializing the MP interpreter (*c.f.* Figure 6.10. The specializer initially unfolds the invocation of `mp-while` on the `(while next ...)` loop on a store where `out` is known to be `(top . nil)`, then replaces the unfolded version with a specialization where `out` is `(top . top)`. This results in the building of two specializations of `mp-while` for the inner `(while kn ...)` loop, one for `out=(top . nil)`, and one for `out=(top . top)`, even though the first such specialization does not appear in the residual program. Thus, without the re-use mechanism, the specializer builds 5 specializations, 4 of which appear in the residual program. Since the re-use mechanism detects that the value of `out` is not used in specializing `mp-while` on either of the `(while kn ...)` loops, it builds only 3 specializations, all of which appear in the residual program.

The amount of redundancy incurred as a result of iteration in the specializer, and the amount of savings realizable from avoiding such redundancy, is highly dependent on the specializer's termination mechanism and on the complexity (height) of its type lattice. For example, under one version of FUSE with a simple type lattice and termination criterion, specializing an 800-line partial evaluator on unknown inputs built 29 specializations, 16 of which appeared in the residual program; adding the re-use criterion avoided the construction of only one specialization, and didn't even save enough time to cover the overhead of doing so, while specializing the same program under a version of FUSE with an expensive termination criterion and a larger type lattice saved a factor of 10 in specialization time.

6.5.2 Future Issues

The examples above show that our re-use mechanism can be worthwhile for current program specializers. We believe that the need for such mechanisms will only increase in the future, as specializers and the languages they operate on become more powerful.

- **Side Effects:** Existing specializers for languages with side effects make little use of information about values in the store; indeed, some, such as [14, 21] force all store

operations to be residualized. Since most procedures access only a small fraction of the cells in the store, once information about values in the store is used, it will be important to base re-use decisions only on the portion of the store actually used in building the specialization (this is the same as the problem of dealing with the store in an interpreter for an imperative language, except that, instead of being reified in the interpreter, the store has been moved into the language, where it must be handled explicitly by the specializer). Some work on imperative languages has encountered this problem; both [44] and [80] limit the use of the store in making re-use decisions.

- **Typed Languages:** The specialization of languages with types, particularly those with polymorphism and subtyping, will lead to more opportunities for redundant specialization. For example, a parametrically polymorphic procedure cannot "use" any part of its polymorphic argument, so such arguments should not be considered when making re-use decisions. Similarly, in an object-oriented language, method dispatch does not always use the exact class of an object, but merely the fact that it is a subclass of some other class, from which it inherited the method. That is, if a specialization is built on an argument of class a, but only invokes the argument on operations (message sends) for which a doesn't override or extend the method it inherits from some superclass b, then the specialization is valid for all objects whose class is a subclass of b, not just objects of class a.

- **Improved BTA:** To date, redundant specialization has not been a major problem for offline specializers because the binding time analyzer limits the amount of information that the specializer considers when building specializations. As we move to more sophisticated binding time analyses, such as polyvariant BTA [83, 22, 76] and BTA which allows the use of known properties of otherwise unknown values [29], more information will be made available at specialization time. It is not clear that all of this information will indeed be useful for performing reductions; thus, we foresee an increased risk of redundant specialization.

We also foresee several uses other than the re-use of specializations for our re-use mechanism. First, with only minor modifications, it should be possible to use our mechanism to share unfolded versions of procedures as well as specialized versions of procedures. This is important when dealing with contination-passing-style code; for example, specializing

```
(lambda (k x y z)
  (if (foo x)
      (k y)
      (k z)))
```

on unknown `x` but known `k`, `y`, and `z` would normally unfold `k` on both `y` and `z`. Often, however, the residual code for both unfoldings is the same. For example, `k` might use its argument in a flow-dependent manner (*i.e.,* only use it when some free variable is true), or it might never use its argument (*i.e.,* just cons it into the final return value). Extending our mechanism to unfoldings is simply a matter of caching argument vectors and computing MGIs for unfoldings as well as for specializations. One simple way to do this is to specialize all calls, then, after specialization, post-unfold all calls with only one call site (such a 1-bit reference counting scheme is used in [14]). Of course, specializing all calls might require too much bookkeeping and caching, so some heuristic for deciding which calls are worth caching might be necessary (Similix-2 [11] chooses to specialize all `if`-expressions and closure bodies). In FUSE, post-unfolding would be quite simple; because of structure sharing in the symbolic value representation of residual code, all that is necessary is to replace the code slots of the symbolic values representing the formals with the code slots of those representing the actuals, and to re-instantiate the return value.

Second, since partial evaluation can be considered to be a form of abstract interpretation [29] (though this interpretation is not necessarily over a finite-height domain), our mechanism might be useful in other abstract interpretation settings. In particular, the control-flow analyses of Shivers [105] and Harrison [50], and the type analysis of Aiken and Murphy [3, 84] must recompute the analysis of a procedure each time its abstract arguments move up in the lattice. If their abstract interpreters were to keep track of which information was actually used to perform abstract reductions during the analysis, they might be able to avoid some amount of recomputation. At this point, we cannot say what level of speedup this might provide, but since some of these analyses are quite time-consuming, re-use mechanisms might be an important part of making them practical.

6.6 Related Work

Existing program specializers use various techniques for achieving re-use of specializations. The class of *monovariant* specializers builds only one specialization of each function definition in the program, thereby achieving re-use at the expense of accuracy. The class of

polyvariant specializers, which build a new specialization for each combination of argument values passed to a function, re-uses specializations only when their argument specifications are the same. This approach gives a more accurate specialization, but, as we have shown, can build redundant specializations.

Such behavior is less evident in polyvariant specializers used in conjunction with a monovariant binding time analysis, because the BTA limits the amount of information that the specializer considers when building specializations. Similarly, specializers with more coarse-grained argument specification models (such as those without "partially static structures" or typed unknown values) are less prone to build redundant specializations because there is less information available to cause the building of such specializations. Because this relative lack of information may adversely affect the quality of specializations, the FUSE strategy attempts to avoid placing *a priori* limitations on information use at specialization time, and instead treats the redundant specialization problem explicitly.

This section describes related work on re-use and limiting of specialization, type systems for specializers and explanation-based generalization upon which we have relied, or which is relevant to an understanding of our work and its place in the field.

6.6.1 Re-use and Limiting of Specializations

In [76][Section 7.3], Launchbury describes a specializer for a statically typed first-order language with parametric polymorphism only. Since all polymorphism is parametric, specializing with respect to polymorphic parameters would result, at best, in the inlining of constants, yielding trivial specializations. Thus, Launchbury's specializer limits itself to building specializations with respect to those portions of the available information over which the function is not polymorphic.

FUSE operates on a dynamically typed language in which both parametric and ad-hoc polymorphism (often over ad-hoc "types" built by user programs) are widely used. Since FUSE attempts to specialize functions only on information which is used to perform non-trivial reductions at specialization time, parametric polymorphism will not give rise to multiple specializations, while ad-hoc polymorphism will. Thus, we obtain the desired result without the need for assumptions about the type system.

Launchbury also suggests (Section 8.2.5 of [76]) using projection-based strictness analysis to deduce that certain information in the arguments to a function will not be used at runtime, and having the specializer treat that function as though it had a smaller argument

domain (*i.e.* without the useless information). The specializer could use this knowledge to eliminate unnecessary runtime parameter passing, and to avoid the construction of redundant specializations based on the useless information. Such a mechanism could be considered a static approximation of FUSE's; its static nature would prevent it from exploiting information that is not available until specialization time. However, from an efficiency standpoint, static analyses are attractive; we are actively investigating a combination of the two.

Gomard and Jones [44] present a specializer for an imperative language, and discuss the problem of redundant specialization due to "dead" variables assuming multiple static values. Their solution consists of a pre-processing phase which marks each program point with the names of the variables known to be live (in the sense of [2]) at that point; only the values of those variables are used in making re-use decisions. Compared with our mechanism, this scheme has two disadvantages. Because the analysis is static, it cannot handle conditional liveness based on specialization-time values. Second, since it operates at the granularity of variables rather than values, it cannot detect redundancy due to partially live structures such as the store in an interpreter.

In [31], Cooper, Hall, and Kennedy define an optimization called *procedure cloning* which has strong similarities to program specialization, and describe a re-use criterion for clones. In this approach, the programmer defines a set of targeted optimizations (*e.g.,* constant propagation) and a domain of interesting values (*e.g.,* for constant propagation, sets of $\langle$*variable name*, *constant value*$\rangle$ pairs). An analysis similar to specialization propagates the values through the program, computing the set of all argument vectors ("cloning vectors") for each procedure. Duplication is limited by the choice of the domain of interesting values, and by filtering the argument vectors to omit values that can't have any affect on the desired optimizations within the procedure or its callees. A second phase merges identical clones using an evaluation function that maps from argument vectors to a vector of values for "interesting" expressions which affect optimizations ("state vector"). The ability to merge clones based on state is interesting. For example, cloning can merge specializations of

```
(lambda (x y a) (* (+ x y) a))
```

for {`x=1`, `y=2`, `a=`$\top_{Number}$}, and {`x=2`, `y=1`, `a=`$\top_{Number}$}, because `(+ x y)` evaluates to `3` in both cases. Since our algorithm views `(+ x y)` as "using" the actual values of `x` and `y`, it cannot perform this merge. This minimization is possible because only a limited subset of the expressions are "interesting;" in a program specializer where all expressions are

"interesting," it would be tantamount to comparing the residual code for two specializations (*i.e.,* instead of comparing specializations on the values of their parameters, compare them on the values of all expressions reduced while computing their bodies). A third phase actually performs the cloning operation, making a copy of each procedure for each different "state vector" until a program size threshold is reached. This algorithm is potentially much more efficient than specialization because computing the "used" information for a clone requires only evaluating the "interesting" expressions, unlike specialization, which requires evaluating all (modulo control flow) expressions in a procedure's body. It is not clear how to extend this technique to program specialization.[18]

The re-use analysis described in [92] and this chapter is eager, in that it treats a value as "used" (*i.e.,* reducing its domain specification) whenever a specialization-time reduction is performed which is limited to that particular value. However, not all computed values affect the building of the specialization. For example, specializing

```
(lambda (x y z)
  (let ((t (+ x y)))
    (if z
        (+ x 1)
        t)))
```

on `x=1`, `y=2`, and `z=#t` uses both `1` and `2` when computing the value of `t`, or `3`. However, since `z` is true, the computation of `t` is dead code, and does not affect the building of the specialization. Similarly, any computations performed in computing a return value are really only necessary if the specialization of the caller makes use of the return value. Katz [70] has proposed performing specialization-time reductions (and corresponding MGI calculations) lazily to avoid this problem.

6.6.2 Types

We divide existing work on type systems for specializers into two categories based on when the type analysis is performed: systems that perform type analysis at specialization time,

[18] If we could decide which expressions were "interesting," we could make the evaluation function explicit in the structure of the source by hoisting "interesting" expressions such as `(+ x y)` of their enclosing `lambda`; then, the parameter `3` would be identical in both cases, and reuse could take place [32]. Of course, this merely delegates the problem to each call site; taken to extremes, this would require hoisting all statically reducible expressions as high as possible.

and systems that perform it in a pre-pass, at which time only the binding times of values are known.

Several online specializers [67, 49, 8, 101, 29] maintain type information at specialization time. REDFUN-2 [49], can also propagate information out of conditionals and from the test of a conditional into its branches, but handles only scalar types (though it does compute disjoint unions, and a limited form of negation, which FUSE doesn't). In certain restricted cases, REDFUN-2 also reasons about values returned by specializations of non-recursive procedures, though it lacks a template mechanism, and thus must compute all return values explicitly. The online systems of Berlin [8] and Schooler [101] propagate information downward using *placeholders* and *partials*, respectively, both of which are similar to FUSE's symbolic values. The parameterized partial evaluation (PPE) framework of Consel and Khoo [29] is a user-extensible type system for program specialization which can infer and maintain "static information" drawn from finite semantic algebras. The online variant of PPE performs generalization to compute return values for `if` expressions, but its behavior with respect to return values of residual calls and parameters to specializations of higher-order procedures is unspecified.

The SIS [42] system uses predicates as specifications, giving finer-grained specifications than FUSE's type specifications, and offers the possibility of using theorem proving at folding time to show that re-use of specializations was proper. Unfortunately, SIS is not automatic: it lacks a theorem prover, and thus leaves all reasoning about generalization and folding to the user. Futamura's "Generalized Partial Computation" [41] also advocates the use of predicates as specifications, and the use of a theorem prover to further specialize the arms of conditionals based on knowledge of the test. No implementation of this scheme has, as yet, been reported in the literature.

The partially static binding time analyses of Mogensen [82] and Consel [23] reason about structured types, including recursive ones. Both operate by building finite tree representations of these data types. Consel's facet analysis [29] adds the ability to deduce that certain properties of unknown values will be known at specialization time. Launchbury's projection-based binding time analysis [76] also models recursive types; it assumes a statically typed language, and constructs a finite domain of approximations from the type declarations. These analyses only produce descriptions of structures whose size will be known at specialization time; since FUSE is an on-line specializer, it doesn't need to build recursive descriptions of such values, but instead simply operates on them. Similarly,

binding time analysis can propagate information out of conditionals only when the test is static, whereas FUSE can do this in both the static and dynamic cases. Several program transformations [83, 26, 55] have been developed to address this problem of offline systems. The techniques used by partially static binding time analyses to represent specialization-time structures at BTA time may have interesting applications in online specialization. Modifying these techniques to run at specialization time (to describe runtime values),would give FUSE the ability to describe structures such as a list of unknown length that contains only integers.

Young and O'Keefe's *type evaluator* [118] is very similar to FUSE, but cannot be considered to be a program specializer because it doesn't build specializations. The type evaluator discovers types (including recursive types) using a variety of techniques, including fixpointing and generalization as used in FUSE. Unlike FUSE, however, the type analysis performed by the type evaluator is monovariant in the sense that a polymorphic formal parameter of a function will be assigned the least upper bound of the types of the corresponding actuals from all calls to the function, while a polyvariant type analysis would be free to build a separate, more accurately typed specialized version of the function for each type of actual parameter.

The FL type inferencer of Aiken and Murphy [84, 3] treats types as sets of expressions rather than sets of values, avoiding some of the difficulties usually encountered when treating function types. To some degree, FUSE uses similar techniques, specializing functions at each call site in order to compute their return types, instead of attempting to build and instantiate type signatures for functions. Our reuse mechanism adds a limited template functionality, but still stops short of computing completely general function types.

6.6.3 Explanation-Based Generalization

The process used by FUSE for computing the most general index of a specialization is similar to the technique known in the machine learning community as Explanation-Based Generalization (EBG). In its usual formulation [81], EBG consists of taking an example and an explanation of the construction of the example, and performing goal regression through the explanation, producing a more general rule. In the case of a specializer, the example is a specialization, the explanation is the trace of reductions performed by the specializer while building the specialization, and the generalized result is the specialization that would be built if the same function were to be specialized on the MGI of the example specialization.

This similarity is easier to see in the alternate formulation of [35], called Explanation-Based Learning (EBL), which avoids goal regression by maintaining two substitutions, SPECIFIC and GENERAL. In the case of FUSE, these substitutions correspond to the value and domain specifications of the index, respectively.

This link between specialization and EBG has been noted before by van Harmelen and Bundy [113], who showed how to convert a partial evaluator into an EBG system by leaving out unifications due to operational predicates (primitive reductions) run by the partial evaluator. However, their notion of a partial evaluator is somewhat simplistic, as it assumes that the specialization can be arrived at by merely unrolling the function description on its inputs, producing a set of leaves of the proof tree (residual applications of primitives) as a result. Their sample partial evaluator does not reason about recursion, and cannot build loops. Thus, the EBG system derived from such a simple partial evaluator can only generalize explanations that are trees, meaning it cannot generalize arbitrary recursive programs. This leaves open the possibility of performing the PE-to-EBG transformation on FUSE, possibly yielding a new class of EBG systems.

6.7 Summary

We have shown that existing program specializers using polyvariant specialization can build redundant specializations. We have formulated a re-use criterion based on the domains of specializations, and have shown how an approximate version of this criterion, based on types, is implemented in our program specializer, FUSE. Adding our re-use algorithm to FUSE not only allowed it to produce residual programs with fewer specialized functions, also allowed it, in some cases, to run more quickly.

We plan to explore several avenues. We plan to add support for recursive datatypes to FUSE, increasing the accuracy of both specialization and re-use. Decreasing the granularity of the analysis could enable the specializer to re-use residual expressions, rather than just specializations, helping to build smaller residual programs. The speed of the specializer could be improved further by allowing it to re-specialize specializations instead of always specializing original function definitions (this is not possible at the moment because FUSE's input and output languages are not the same). Efficiency could also be gained by developing a version of our mechanism which works under offline specializers, which can be self-applied. More work remains to be done in the area of limiting specializations; perhaps a cost-based

model, such as that used by some code generation strategies, could be used. Finally, we hope to explore the use of our specializer in explanation-based learning.

Chapter 7

Program Generator Generation

In this chapter, we turn away from specialization mechanisms designed to increase the efficiency of specialized programs, and concentrate instead improving the efficiency of the specialization process. One common technique for improving the speed of the specialization of a particular program is to specialize the specializer itself on that program, creating a specializer, or *program generator*, customized for that particular program.

Much research has been devoted to the problem of generating *efficient* program generators, which do not perform reductions at program generation time which could instead have been performed when the program generator was constructed. The conventional wisdom holds that only *offline* program specializers, which use *binding time annotations*, can be specialized into such efficient program generators. In this chapter, we argue that this is not the case, and demonstrate that the specialization of a nontrivial *online* program specializer similar to the original "naive MIX" can indeed yield an efficient program generator. We will not, however, demonstrate "self-application;" although FUSE is the first specializer sufficiently powerful to specialize "naive MIX," it is not sufficiently powerful to specialize itself.

The key to our argument is that, while the use of *binding time information* at program generator generation time is necessary for the construction of an efficient custom specializer, the use of explicit *binding time approximation* techniques is not. This allows us to distinguish the problem at hand (*i.e.,* the use of binding time information during program generator generation) from particular solutions to that problem (*i.e.,* offline specialization). We show that, given a careful choice of specializer data structures, and sufficiently powerful specialization techniques, binding time information can be inferred and utilized without

the use of explicit binding time approximation techniques. This allows the construction of efficient, optimizing program generators from online program specializers.

This chapter has seven sections. We begin with an introduction to program generator generation. Section 2 describes a small online specializer, TINY, which will be used in subsequent examples. In Section 3, we demonstrate a naive approach to program generator generation, the inefficiency of the resultant program generators, and our solution to this problem using FUSE. The fourth section describes several extensions to TINY, and how they affect the problem of program generator generation. Section 5 describes related work in efficient program specialization, using both offline and online techniques. We conclude with several directions for future work in efficient online specialization.

7.1 Introduction

Program specialization is useful when some of a program's inputs remain constant over several executions of the program. If we specialize the program on those constant values, the specialized program can be applied repeatedly to the non-constant values, allowing the cost of specialization to be amortized across the repeated executions of the specialized program. For example, instead of repeatedly executing an interpreter on the same program, but different inputs:

```
result-1 := interpreter(program-1,inputs-1)
result-2 := interpreter(program-1,inputs-2)
...
result-3 := interpreter(program-1,inputs-n)
```

we can specialize the interpreter on the program, and use the specialized program several times. Assuming the existence of a two-input procedure, `specializer`, which takes a program and a description of its constant inputs, and produces a program specialized on those inputs, we could instead execute the sequence

```
specialized-interpreter := specializer(interpreter,program-1)
result-1 := specialized-interpreter(inputs-1)
result-2 := specialized-interpreter(inputs-2)
...
result-3 := specialized-interpreter(inputs-n)
```

The number of executions necessary to repay the cost of specialization depends upon the degree to which specialization improves the speed of the program, and upon the cost

of running the specializer. The quality of specialization has been improved by various techniques, while the efficiency of specialization has been addressed primarily by means of program generator generation; that is, specializing the specializer itself.[1]

The idea of improving the efficiency of specialization by specializing the specializer, independently discovered by Futamura [40] and Ershov [39], is based on the same observation we made above. That is, if a program is executed repeatedly on a constant input, we can benefit by specializing the program on that constant input, and executing the specialized program instead. In many cases, the specializer itself is executed repeatedly on a constant input, namely the program to be specialized. For example, an interpreter may be specialized on different programs, as in

```
specialized-interpreter-1 := specializer(interpreter-1,program-1)
specialized-interpreter-2 := specializer(interpreter-1,program-2)
...
specialized-interpreter-n := specializer(interpreter-1,program-3)
```

which can be replaced by the sequence

```
specialized-specializer := specializer(specializer,interpreter-1)[2]
specialized-interpreter-1 := specialized-specializer(program-1)
specialized-interpreter-2 := specialized-specializer(program-2)
...
specialized-interpreter-3 := specialized-specializer(program-n)
```

which is more efficient. This use of the specializer to improve itself is called *self-application*, and the specialized specializer is often referred to as a *program generator*. The generation of efficient program generators has motivated much of the recent research in program specialization.

All that is required for self-application is that the specializer be written in the same language as the programs it processes, so that it can treat itself as a program to be specialized. This *autoprojection* property, although sufficient to allow self-application to be performed, is insufficient to guarantee efficient results. That is, when the specialized specializer runs, it

[1] Several other approaches include handwriting a specializer generator [49, 56] and performing more operations statically prior to specialization time [25]. This chapter addresses only the specializer specialization technique.

[2] Of course, we know more about the specializer's inputs than this. Not only do we know that the program to be specialized is `interpreter-1`, we also know that the interpreter will be specialized with its first argument (the program) known and its second arguments (the inputs) unknown. As we shall see in Section 7.3.1, failure to make use of this additional information will result in an inefficient `specialized-specializer`.

may perform reductions which instead could have been performed once, when the specializer was specialized.

This desire to construct efficient program generators motivated the invention of *offline* program specialization[64], in which all of the specializer's reduce/residualize decisions are made prior to specialization time, usually via an automatic prepass called Binding Time Analysis (BTA). The results of these decisions are made available to the specializer as annotations on the source program. Because most of the specializer's behavior[3] is determined by the source program and the binding time annotations, both of which are supplied as part of the known (constant) input when the specializer is specialized, much of the specializer's behavior can be determined (and the corresponding reductions performed) at self-application time. In particular, the resulting program generator will contain no code to examine the binding times of any of its inputs or to make reduce/residualize decisions. The residual program will of course contain code to perform reductions and construct residual expressions, but the reduce/residualize pattern will be fixed. In the case of specializing the specializer on an interpreter, the resultant program generator bears a similarity to a compiler, in that it makes reductions based on the "static semantics" (syntactic dispatch, static environment lookup, static typing) [27] of the program and constructs residual code to implement the "dynamic semantics" (dynamic typing, store operations, primitive reductions) of the program. No unnecessary comparisons are performed; the program generator "knows" that the information necessary to reduce the static semantics will be present, and that the information needed to reduce the dynamic semantics will not be present, because that information (the "program division" of [62]) was explicitly available when the program generator was constructed.

There is a tradeoff here between the efficiency of the program generator and the the quality of the programs it produces. The program generator's efficiency depends on making all of the specializer's reduce/residualize decisions reducible at program generator generation time, which is accomplished by "hoisting" them out of the specializer into the BTA prepass. However, this "hoisting" is, by definition, approximate, since some binding time information is unavailable at the time the BTA runs (*c.f.* Chapter 2). Thus, specializing an offline specializer yields an efficient generator of potentially inefficient specialized programs;

[3]This includes syntactic dispatch, environment lookup, and reduce/residualize choices, but not the actual results of reductions, since the input values for the reductions are determined by the input values to the specializer.

making all reduce/residualize decisions at program generator generation time means that, at program generation time, some performable reductions may not be performed.

Online program specializers are distinguished by their willingness to make some reduce/residualize choices at specialization time. This makes them inherently slower than their offline counterparts, due to the need to represent unknown values explicitly, and to make more complex decisions at specialization time (rather than just consulting precomputed annotations). However, because the specializer's reduce/residualize decisions are based on specialization-time information, rather than on BTA-time approximations to specialization-time information as in offline specializers, better choices can be made, resulting in smaller, faster residual programs. This desire for better residual programs has motivated much of the research in online specialization (*c.f.* Chapter 2). If one could construct an efficient program generator by specializing an online specializer, it would not be as fast as a program generator constructed from an offline specializer, because it would still make some of its decisions at program generation time. In the case of specializing the specializer on an interpreter, we would expect the following of the resultant program generator: reductions relating to the static semantics of the program should be performed without examination of the binding times of the program text or interpreter data structures used to evaluate those semantics, since the static nature of those structures should be deducible at program generator generation time, while operations relating to the dynamic semantics may conditionally be reduced *or* residualized, depending on the availability of sufficient information to reduce them. In other words, a program generator constructed from an online specializer is an "optimizing compiler" which is willing to reduce some of the dynamic semantics of the program, rather than a "non-optimizing compiler" which assumes that none of the dynamic semantics will be reducible.

Both offline and online specializers have been successfully specialized, but, to date, only offline specializers and online specializers using offline-style binding time approximation techniques[4] have yielded efficient program generators. Indeed, the conventional wisdom holds that explicit binding time approximations are essential to efficient program generation. This chapter demonstrates that, although such methods are indeed effective for efficient program generation, they are not essential. We will demonstrate that the specialization of a nontrivial *online* program specializer without BTA techniques can indeed yield an

[4]The "specialization-time BTA" approach of Glück has indeed yielded an efficient, though non-optimizing, program generator. Section 7.5.2 treats Glück's work in more detail.

efficient, accurate program generator. Our solution will require not only a careful choice of specializer data structures, but also a particularly powerful specializer to ensure that the information in those data structures is not prematurely lost (generalized) at program generator generation time. Because of the complexity of such a specializer, we will not demonstrate full self-application; instead, we will show that a nontrivial online program specializer with power at least equivalent to that of MIX [64] and Schism [22] can yield an efficient program generator when specialized by FUSE. We will specialize our small online specializer on several programs, and will evaluate the efficiency of the results.

A note on terminology: Thus far in this dissertation, we have used the terms "known" and "unknown" when describing specialization-time values, and have used the terms "static" and "dynamic" only to describe the approximations to "known" and "unknown" made by offline systems. Because we expect that the reader may wish to contrast our account of specializer specialization (Section 7.3) with other treatments of the subject (such as [15, 12, 83]), we adopt the standard offline terminology *for this chapter only*. That is, a value is "static" if it is known at specialization time, and "dynamic" otherwise, without any connotation of approximation.

7.2 TINY: A Small Online Specializer

This section describes TINY, a small but nontrivial online program specializer for a functional subset of Scheme [88] which we will use to demonstrate the construction of online program generators. For reasons of clarity, we will first describe a "watered-down" version of TINY using a denotational-semantics-like language, then describe the actual Scheme implementation. This will allow us to use the abbreviated description in many of the examples in later sections, dropping down into the implementation only when necessary.

The version of TINY we will describe here specializes programs written in a first-order, functional subset of Scheme. A program is expressed as a single `letrec` expression whose body is the name of the goal procedure to be specialized. Both scalars and pairs are supported, but there are no vectors, and no support for partially static structures; that is, any pair containing a dynamic is considered to be dynamic.

7.2.1 Abstract Description

In this section, we will consider a fragment of TINY which partially evaluates a Scheme expression in an environment mapping Scheme identifiers to specialization-time values, returning a specialization-time value. We are primarily interested in how TINY makes reduce/residualize decisions, so that we can examine whether these decisions can be made at the time TINY is specialized. In a first-order language, the interesting reduce/residualize decisions are at conditionals and primitive applications; we will ignore (for now) how the specializer makes generalization decisions, and how it creates, caches and re-uses specializations of user functions.

For brevity, our description will be couched in a denotational-semantics-like language similar to that used in [12], with double brackets around Scheme syntactic objects. Injection and projection functions for sum domains will be omitted. Of course, the real specializer is written in Scheme and operates on (preprocessed) Scheme programs.

TINY represents each specialization-time values as a *pe-value* (an element of the domain *PEVal*), as shown in Figure 7.1. Static values (those known at specialization time) are represented as Scheme values, while dynamic values (those unknown at specialization time) are represented as source language expressions. In the latter case, the expression will compute the runtime value(s) of the specialization-time value. We assume the existence of some helper functions for looking up identifiers in an environment and for coercing values to constant expressions. The fragment of TINY shown in Figure 7.2 specializes expressions. The function *PE* takes a Scheme expression and an environment mapping each Scheme identifier to a *pe-value*, and returns a *pe-value*.

TINY's online nature can easily be seen in the code for processing `if` expressions. After partially evaluating the test expression, e_1, the specializer tests the resultant *pe-value*, p_1, to see if it is a value (*i.e.,* static). If the value is static, it is used to specialize one of the two arms; otherwise, both arms are specialized and a residual `if` expression is returned. Similarly, the code for processing `car` and `cdr` expressions tests the *pe-values* obtained by partially evaluating the argument expressions. If we construct a program generator by specializing TINY, we would like to eliminate not only the syntactic dispatch on the first argument of *PE*, but also as many of these ($p \in Val$) tests as possible. This elimination is the topic of Section 7.3.

Domains

e	$\in$	Exp	*expressions (source and residual)*
x	$\in$	Id	*identifiers*
v	$\in$	Val	*scheme denotable values*
p	$\in$	$PEVal = Val + Exp$	*specialization-time values*
env	$\in$	$Env = Var \to PEVal$	*specialization-time environments*

Function Signatures

PE	:	$Exp \to Env \to PEVal$	*partially evaluate an expression*
$lookup$	:	$Id \to Env \to PEVal$	*look up an identifier*
$resid$	:	$PEVal \to Exp$	*coerce a value to an expression*
car	:	$Val \to Val$	*primitive*
$cons$	:	$Val \to Val \to Val$	*primitive*
$\vdots$			

Figure 7.1: Domains and Function Signatures for TINY

$$
\begin{aligned}
&PE[\![\texttt{(quote } v\texttt{)}]\!]\, env &&= v\\
&PE[\![\texttt{x}]\!]\, env &&= lookup[\![\texttt{x}]\!]\, env\\
&PE[\![\texttt{(if } e_1\ e_2\ e_3\texttt{)}]\!]\, env &&= let\ p_1 = PE[\![e_1]\!]\, env\ in\\
&&&\quad p_1 \in Val \to\\
&&&\quad\ (p_1 = true \to PE[\![e_2]\!]\, env,\ PE[\![e_3]\!]\, env),\\
&&&\quad\ let\ p_2 = PE[\![e_2]\!]\, env\\
&&&\quad\quad p_3 = PE[\![e_3]\!]\, env\\
&&&\quad\quad in\ [\![\texttt{(if } p_1\ (resid\ p_2)\ (resid\ p_3)\texttt{)}]\!]\\
&PE[\![\texttt{(car } e_1\texttt{)}]\!]\, env &&= let\ p_1 = PE[\![e_1]\!]\, env\ in\\
&&&\quad p_1 \in Val \to (car\ p_1),\ [\![\texttt{(car } p_1\texttt{)}]\!]\\
&PE[\![\texttt{(cons } e_1\ e_2\texttt{)}]\!]\, env &&= let\ p_1 = PE[\![e_1]\!]\, env\\
&&&\quad\ p_2 = PE[\![e_2]\!]\, env\\
&&&\quad\ in\ (p_1 \in Val) \wedge (p_2 \in Val) \to\\
&&&\quad\quad (cons\ p_1\ p_2),\\
&&&\quad\quad [\![\texttt{(cons } (resid\ p_2)\ (resid\ p_3)\texttt{)}]\!]\\
&\vdots\\
&resid\ p = p \in Val \to [\![\texttt{(quote } p\texttt{)}]\!],\ p
\end{aligned}
$$

Figure 7.2: Fragment of TINY

7.2.2 Implementation Description

The abstract description of TINY omits details relating to concrete data representations, function application, construction and caching of function specializations, and termination. Because these details affect program generator generation, we address them briefly here.

Data Representations

TINY represents source and residual expressions as abstract syntax trees, which are uninteresting relative to our discussion. The choice of a representation for specialization-time values, namely the *pe-values*, is important. The type *PEVal* is a disjoint union of Scheme values and expressions; one obvious way to implement it is with a tagged record, namely either `(static <value>)` or `(dynamic <expression>)`. Many online specializers, such as the one of [15] and the simple online partial evaluation semantics of [29] capitalize on a relationship between expressions and values, namely, that constant expressions of the form `(quote <value>)` are capable of representing values. Thus, they can omit the tag, and can check the "static-ness" of a value merely by checking to see if it is a pair whose `car` is the symbol `quote`. We will see later (*c.f.* Section 7.3.2) that this optimization makes efficient program generator generation far more difficult; thus, TINY does not use it.

(We're still being a bit dishonest here: TINY doesn't actually implement a *pe-value* as a tag followed by a value *or* a residual code expression, it implements it as a tag followed by a value *and* a residual code expression, similar to the *symbolic value* objects of FUSE [114, 115]. This is necessary if partially static values are to be allowed, since the description of the value (*e.g.,* a pair whose `car` is 4 and and whose `cdr` is unknown) may not be deducible from its residual code (*e.g.,* `(cdr (foo x))`). Since this additional mechanism is necessary only for code generation, and not for making reductions, we can, without loss of generality, ignore it for the remainder of the chapter. We presented it here only so that the descriptions of partially static encodings in Section 7.4.2, which also omit code generation details, won't seem strange.)

Function Application

The description in Figure 7.2 doesn't describe the treatment of user functions. Basically, all that needs to be done is to add an extra parameter representing a set of

(*function name*, *function definition*) pairs to the semantics, and to have the semantic function implementing function application look up the function name in this set. After the appropriate formal/actual bindings are added to the environment, the unfolded/specialized body can be constructed via a recursive invocation of the specializer (*i.e.*, a call to the valuation function *PE* of Figure 7.2).

Specialization

TINY is a polyvariant program point specializer [16, 62]; that is, it constructs specializations of certain program points (in this case, user function applications) and caches them for potential re-use at other program points (function applications with equivalent argument vectors). TINY builds specializations in a depth-first manner; it does this by adding a single-threaded cache parameter to the semantics, and posting "pending" and "completed" entries to the cache for each specialization before and after it is completed, respectively.[5]. When the specialization process is complete, the cache will contain definitions for the specialization of the goal function, and any specialization it may (transitively) invoke. The code generator uses these definitions to construct the residual program.

Termination

To terminate, TINY needs to build a finite number of specializations, each of finite size. The latter can be achieved by limiting the amount of unfolding performed, while the former requires that specializations be constructed only on a finite number of different argument vectors. Most online specializers implement these restrictions using a combination of static annotations (finiteness assertions and argument abstraction) and dynamic reasoning (call stacks, induction detection, argument generalization, and explicit filters). TINY uses static annotations for both types of restrictions. Each user function is tagged with a flag specifying whether it is to be unfolded or specialized, while each formal parameter is tagged with a flag specifying whether it should be abstracted to "dynamic" before specialization is performed. Since much of the power of online specialization derives from its use of online generalization rather than static argument abstraction, this might make TINY appear overly simplistic. This is not the case, as adding online termination mechanisms (at least simple ones) does not appreciably increase the difficulty in obtaining an efficient program generator.

[5] For a more formal description of a single-threaded cache, see [29].

```
(lambda (names values) (cons (car names) (car values)))
```

Figure 7.3: A fragment of a hypothetical interpreter

In Section 7.4.1, we will show that is the case.

7.2.3 Example

In this section, we show two examples of TINY in action. The first example shows the specialization of a very small fragment of a hypothetical interpreter, which is rather small and unrealistic, but will be useful later as an example for program generator generation. Our second example shows the specialization of an interpreter for a small imperative language.

Interpreter fragment example

Consider an interpreter for a small imperative language which maintains a store represented as two parallel lists: `names`, which holds a list of the identifiers in the program being interpreted, and `values`, which holds a list of the values bound to those identifiers. Assume that the interpreter contains an expression of the form `(cons (car names) (car values))`; this might be part of a routine to construct an association list representation of the store to be used as the final output of the interpreter. (The expression `(cons (car names) (car values))` is a standard example; we are following the treatment of [15, 12, 43]. The real purpose of such an expression is irrelevant; all that is important is that the binding times of `names` and `values` differ at the time the interpreter is specialized.)

When the interpreter is specialized on a known program but unknown arguments, the list `names`, which is derived from the program, will be static, but the list `values`, which is derived from the arguments, will be dynamic. Thus, the specializer will generate a residual constant expression for `(car names)`, and residual primitive operations for `(car values)` and `(cons (car names) (car values))`.

Rather than examining the entire interpreter, we will abstract this small fragment into a separate program (Figure 7.3). We can use TINY to specialize this program on `names=(a b c)` and `values` unknown by executing the form

```
(tiny fragment-program (list (make-static-peval '(a b c))
                             (make-dynamic-peval)))
```

TINY returns a residual program expressed as a cache; after simple postprocessing (not including dead parameter removal or arity raising), we obtain

```
(lambda (names values) (cons 'a (car values)))
```

which is what we expected. We will return to this example in Section 7.3, where we will generate a program generator for this fragment by specializing TINY on the fragment and unknown inputs.

MP interpreter example

In this chapter, we are not particularly concerned with the efficiency of the specializations constructed by TINY; Instead, we will concern ourselves with the efficiency of the specialization process itself, and the gains to be obtained by program generator generation. However, we would like to show that TINY is indeed a realistic program specializer. To this end, we will show the operation of TINY on an interpreter for a small imperative language.

The MP language [102] is a small imperative language with `if` and `while` control structures, which has become the "canonical" interpreter example for specializers. Figure 7.4 shows an interpreter for this language; it traverses the MP program's commands and expressions in a straightforward recursive-descent manner, while passing a single-threaded representation of the program's store. Because TINY doesn't handle partially static structures, the interpreter represents the store as two parallel lists, one (static) list for the names, and another (potentially dynamic) list for the values.

When we specialize the MP interpreter on a known program and an unknown input, we expect that all reductions depending solely on the static data (*i.e.,* on the program text) will be performed. Thus, all syntactic dispatch should be eliminated, while store operations should be implemented as open-coded tuple operations. This is exactly what happens; if we specialize an MP program to compare the lengths of two lists:

```
(letrec
    ((mp (lambda (program input)
           (let ((parms (cdr (cadr program)))
                 (vars (cdr (caddr program)))
                 (main-block (cadddr program)))
             (mp-command main-block (init-names parms vars) (init-vals parms vars input)))))

     (init-names (lambda (parms vars)
                   (if (null? parms)
                       (if (null? vars)
                           '()
                           (cons (car vars) (init-names parms (cdr vars))))
                       (cons (car parms) (init-names (cdr parms) vars)))))

     (init-vals (lambda (parms vars input)
                  (if (null? parms)
                      (if (null? vars)
                          '()
                          (cons '() (init-vals parms (cdr vars) input)))
                      (cons (car input) (init-vals (cdr parms) vars (cdr input))))))

     (mp-command (lambda (com names vals)
                   (let ((token (car com)) (rest (cdr com)))
                     (cond
                      ((eq? token ':=)
                       (let ((new-value (mp-exp (cadr rest) names vals)))
                         (update names vals (car rest) new-value)))
                      ((eq? token 'if)
                       (if (mp-exp (car rest) names vals)
                           (mp-command (cadr rest) names vals)
                           (mp-command (caddr rest) names vals)))
                      ((eq? token 'while) (mp-while com names vals))
                      ((eq? token 'begin)
                       (mp-begin rest names vals)))))))

     (mp-begin (lambda (coms names vals)
                 (if (null? coms)
                     vals
                     (mp-begin (cdr coms) names (mp-command (car coms) names vals)))))

     (mp-while (lambda (com names vals)
                 (if (mp-exp (cadr com) names vals)
                     (mp-while com names (mp-command (caddr com) names vals))
                     vals)))

     (mp-exp (lambda (exp names vals)
               (if (symbol? exp)
                   (lookup exp names vals)
                   (let ((token (car exp))
                         (rest (cdr exp)))
                     (cond ((eq? token 'quote) (car rest))
                           ((eq? token 'car) (car (mp-exp (car rest) names vals)))
                           ((eq? token 'cdr) (cdr (mp-exp (car rest) names vals)))
                           ((eq? token 'atom) (not (pair? (mp-exp (car rest) names vals))))
                           ((eq? token 'cons)
                            (cons (mp-exp (car rest) names vals)
                                  (mp-exp (cadr rest) names vals)))
                           ((eq? token 'equal)
                            (equal? (mp-exp (car rest) names vals)
                                    (mp-exp (cadr rest) names vals)))))))))

     (update (lambda (names vals var val)
               (let ((binding (car names)))
                 (if (eq? binding var)
                     (cons val (cdr vals))
                     (cons (car vals) (update (cdr names) (cdr vals) var val))))))

     (lookup (lambda (var names vals)
               (let ((binding (car names)))
                 (if (eq? binding var)
                     (car vals)
                     (lookup var (cdr names) (cdr vals)))))))
  mp)
```

Figure 7.4: Interpreter for MP

```
(letrec
 ((mp-while3
   (lambda
    (vals)
    (if
     (caddr vals)
     (mp-while3
      (cond
       ((car vals)
        (if
         (cadr vals)
         (cons (cdar vals) (let ((T44 (cdr vals))) (cons (cdar T44) (cdr T44))))
         (cons
          (car vals)
          (let ((T68 (cdr vals))) (list* (car T68) '() 'a (cdddr T68))))))
       ((cadr vals)
        (cons
         (car vals)
         (let ((T93 (cdr vals))) (list* (car T93) '() 'b (cdddr T93)))))
       (else
        (cons
         (car vals)
         (let ((T119 (cdr vals))) (list* (car T119) '() 'ab (cdddr T119)))))))
     vals)))
  (mp2
   (lambda
    (input)
    (mp-while3
     (let
       ((T22 (list* (car input) (cadr input) '(() ()))))
       (cons
        (car T22)
        (let ((T23 (cdr T22))) (list* (car T23) '#T (cddr T23)))))))))
 mp2)
```

Figure 7.5: Specialized MP interpreter obtained using TINY

```
(program (pars a b) (dec flag out)
  (begin
    (:= flag '#t)
    (while flag
      (if a
          (if b
              (begin (:= a (cdr a))
                     (:= b (cdr b)))
              (begin (:= out 'a)
                     (:= flag '())))
          (if b
              (begin (:= out 'b) (:= flag '()))
              (begin (:= out 'ab) (:= flag '()))))))))
```

we obtain a specialized interpreter containing only operations pertaining to the values of the identifiers in the program (Figure 7.5); all syntax and name list comparison operations have been reduced. Under interpreted MIT Scheme, executing the specialized interpreter is approximately 6 times faster than executing the original interpreter on the comparison program. Larger MP programs (or larger inputs to them) produce better speedups. In this chapter, we are not particularly concerned with the benefits of specializing interpreters, which are well understood. From now on, we will concentrate only on the efficiency of specializations *of* TINY, rather than on specializations constructed *by* TINY.

7.3 Program Generator Generation

Having described our simple online partial evaluator, we now turn our attention to producing program generators by specializing it. As we shall soon see (*c.f.* Section 7.3.2), TINY is insufficiently powerful to produce an efficient program generator when self-applied; constructing an efficient program generator from TINY will require the use of a more powerful specializer. By the end of this section, we will know how powerful that specializer must be; for now, we assume the existence of a procedure `specialize`, which takes as arguments the Scheme program to be specialized, and the argument values on which it is to be specialized. To avoid concerning ourselves with this (hypothetical) specializer's representations, we will assume (for now) that, for input purposes, it uses ordinary Scheme values for static values, and `<dynamic>` for dynamic values.

We begin by demonstrating the specialization of TINY on a known program and a completely unknown input specification. We will see that, because this approach fails to specify the binding times of the elements of the input specification, it generates an overly general program generator. The remainder of this section will describe how the binding time information can be represented and maintained, and will give several examples of the generation of efficient program generators from TINY.

7.3.1 The problem of excessive generality

We will begin by considering the interpreter fragment from Section 7.2.3. We can accomplish this by specializing TINY on the interpreter fragment and a dynamic[6] argument list, as in

```
(specialize tiny-program fragment-program <dynamic>).
```

Consider what happens as TINY is specialized. At best, the specializer can execute all of those operations in TINY's implementation which depend solely on its program input, and on constants in TINY itself. Referring to the description of Figure 7.2, we can see that the syntax dispatch (all matching of arguments in [[]] brackets) can be eliminated. Also, if `specialize` implements partially static structures (or if TINY's implementation maintains the specialization-time environment as two lists, one for the names and one for the values) environment accesses can be reduced to fixed chains of tuple accesses (allowing for other optimizations such as arity raising). However, none of the static/dynamic tests (*i.e.,* those of the form $p_1 \in Val$) or any of the primitive operations (*i.e.,* $(p_1 = true)$, $(car\ p_1)$, or $(cdr\ p_1)$) can be reduced. Of the semantic parameters omitted in Figure 7.2, the table of function definitions is available, and thus user function lookups can be reduced, but the specialization cache is dynamic, and thus all cache lookups remain residual. An abstract fragment of the resultant program generator is shown in Figure 7.6. To indicate calls to a specialized version of a semantic function, we subscript the function name with the argument on which it was specialized.

Code of this form is obtained when TINY is self-applied; using a more powerful specializer to specialize TINY on this program does not provide any additional improvement because the necessary binding time information is simply not available.

[6] In this instance, "dynamic" means unknown at program generator *generation* time, not at program generation time. That is, the specializer used to specialize TINY knows nothing about the value passed as TINY's second actual parameter.

$$
\begin{aligned}
PE_{\texttt{(cons (car names) (car values))}}\, env = {} & \mathit{let}\ p_1' = \mathit{let}\ p_1 = \mathit{lookup}_{\texttt{names}}\, env\ \mathit{in} \\
& \qquad p_1 \in \mathit{Val} \rightarrow (\mathit{car}\ p_1),\ [\![\texttt{(car}\ p_1\texttt{)}]\!] \\
& \quad p_2' = \mathit{let}\ p_2 = \mathit{lookup}_{\texttt{values}}\, env\ \mathit{in} \\
& \qquad p_2 \in \mathit{Val} \rightarrow (\mathit{car}\ p_2),\ [\![\texttt{(car}\ p_2\texttt{)}]\!] \\
& \mathit{in}\ (p_1' \in \mathit{Val}) \wedge (p_2' \in \mathit{Val}) \rightarrow \\
& \qquad (\mathit{cons}\ p_1'\ p_2'), \\
& \qquad \mathit{let}\ p_1'' = p_1' \in \mathit{Val} \rightarrow [\![\texttt{(quote}\ p_1'\texttt{)}]\!],\ p_1' \\
& \qquad\quad p_2'' = p_2' \in \mathit{Val} \rightarrow [\![\texttt{(quote}\ p_2'\texttt{)}]\!],\ p_2' \\
& \qquad \mathit{in}\ [\![\texttt{(cons}\ p_1''\ p_2''\texttt{)}]\!]
\end{aligned}
$$

Figure 7.6: Fragment of overly general program generator constructed from TINY

The program generator of Figure 7.6 is similar to that obtained by the DIKU researchers [15, 12] when self-applying a simple online specializer on an interpreter. We can view a program generator constructed from an interpreter as a "compiler," since it maps a program written in the language implemented by the interpreter (perhaps a small imperative language; let's call it L) into the language used to implement the interpreter (Scheme).[7] Unfortunately, our program generator isn't a very efficient compiler; it's overly general. In particular, it doesn't know that, at compilation time, the program input will always be static, and the data input will always be dynamic. That is, both

```
(program-generator L-program <dynamic>)
```

and

```
(program-generator <dynamic> L-inputs)
```

are perfectly legal invocations of the program generator. The first takes an L-program and returns a Scheme program which maps a list of inputs to the L-program into the output of the L-program; this is what we commonly think of as compilation. The second takes a list of inputs and returns a Scheme program which takes an L-program and applies it to those

[7] Of course, this isn't really the case. Much of the complexity of "real" compilers lies in dealing with resource allocation issues, such as register allocation, memory management, and the like, which have not been addressed by interpreter-based program generation techniques because it's difficult to expose such issues in an interpreter.

inputs; this isn't particularly useful since very few (if any) expressions in the interpreter depend on the inputs alone, meaning that the program returned by the program generator is unlikely to be faster than the original interpreter for L. Indeed, our program generator is not a compiler, but instead a specializer specialized on a particular *program* (the interpreter) but not on any particular *argument vector* for that program. Thus, the program generator *must*, by definition, be prepared to accept *any* argument vector, be its elements static or dynamic.

In other words, we got what we asked for; we just asked for the wrong thing. How can we remedy this situation? The answer is that if we want the program generator to have, "built-in," certain assumptions about the binding times of the arguments to the interpreter, we must provide that binding time information to TINY at the time the program generator is constructed. We know of two ways of providing this information:

1. **As binding time annotations on the L-interpreter:** the L-interpreter is augmented with a set of annotations which specify the binding times of each expression in the interpreter; the specializer uses these to make its reduce/residualize decisions. Since the interpreter source (and its binding time annotations) are available when the specializer is specialized, all computations depending solely on the binding time annotations are reduced, and the resultant program generator contains no binding time computations whatsoever. This is the major rationale behind the development of offline specialization techniques [64, 15, 12, 83]. Offline specialization solves the generality problem with relatively little added mechanism in the specializer (indeed, offline specializers are usually smaller than their online counterparts, since specialization-time values no longer need be tagged). Unfortunately, the need to compute the binding time annotations prior to specialization time introduces certain inaccuracies and makes certain optimizations difficult, if not impossible (*c.f.* Chapter 2).

2. **As part of the arguments on which TINY specializes the L-interpreter:** instead of specifying `<dynamic>` as the type of TINY's second argument (the argument vector to the L-interpreter), specify the binding time of each argument, leaving the values dynamic. That is, instead of executing

```
(specialize tiny-program fragment-program <dynamic>).
```

to construct the program generator, instead use[8]

```
(define names (make-static-peval <dynamic>))
(define values (make-dynamic-peval))
(specialize tiny-program fragment-program (list names values))
```

which indicates that the `names` argument to `fragment` will be static and the `values` argument will be dynamic at program generation time (when the specialized version of TINY runs). At program generator generation time (when `specialize` runs), we don't know what the static value of the `names` argument to `fragment` *is*; we only know that it's static. Thus, the *pe-value* on which TINY is specialized contains a tag of `static` but a value of `<dynamic>`. In effect, the values on which TINY is symbolically executed by `specialize` mirror those on which TINY was executed in the invocation of TINY on `fragment-program` on page 222, except that the value attribute of the *pe-value* representing the static first argument (*i.e.,* `'(a b c)`) has been replaced by an attribute which is dynamic at program generator generation time, but which will be known when the specialized version of TINY runs (*i.e.,* `<dynamic>`).

This solution also produces the desired result, but without the conceptual overhead or accuracy limitations of binding time annotations. It does, however, require significant additional complexity in the specializer used to construct the program generator (our hypothetical `specialize`), relative to TINY or to offline specializers.[9] The remainder of this section describes these additional requirements on the specializer.

7.3.2 Program generator generation without binding time approximations

The second solution to the problem of an excessively general program generator worked by specifying the binding times of the *pe-values* passed to TINY, but leaving the values of the *pe-values* unspecified. This, in and of itself, is not sufficient to assure that the resultant

[8] Note that `make-static-peval` and `make-dynamic-peval` are abstractions specific to TINY, not to `specialize`. We are embedding the meta-value `<dynamic>`, which is an abstraction of `specialize`, in the value slot of a TINY *pe-value*.

[9] If self-application is desired, then these complexity requirements also apply to the specializer being specialized.

program generator will contain no unnecessary binding-time-related reductions. There are two issues which must be addressed:

- Representing the binding time information, and
- Preserving the binding time information

We will deal with each of these issues in turn.

Representing binding time information

The essence of our method is that TINY's *pe-value* objects represent both a binding time and a value or residual code expression. By embedding a (specializer specialization time) dynamic value inside a (specialization time) *pe-value*, we can communicate the binding time information attribute of the *pe-value* without being forced to specify the value/expression attribute.

One consequence of this encoding scheme is that the specializer (`specialize`) used to construct the program generator must be able to represent partially static values. That is, it must be able to represent a TINY *pe-value* of the form `(static <dynamic>)` (*i.e.,* a list whose first element is the symbol `static` and whose second element is unknown). Without partially static structures, `specialize` would represent such a *pe-value* as `<dynamic>`, at which point the binding time information would be lost, and we would once again obtain an overly general program generator. It might at first appear that we could use a binding time separation technique, similar to the parallel name/value lists used to represent stores in interpreters. That is, we could separate the tag and value/expression fields of a *pe-value* into two separate values, so that the dynamic nature of the values at program generator generation time won't pollute the static tags. This is, in effect, what is performed by offline partial evaluation strategies, which separate binding time tags from the input values, attaching them to the program instead.

The problem with such a "separation" approach is that it only works if all of the binding time tags are static. Under a non-partially-static specializer, as soon as a single binding time tag becomes `<dynamic>` at program generator generation time, the entire binding time environment will be seen as dynamic, and all binding-time-related reductions will be delayed until program generation time. Since some binding times cannot be fully determined until the specialized TINY runs (*e.g.,* values returned out of static conditionals with one

dynamic arm, values produced by online generalization, values returned from primitives which perform algebraic optimizations, etc; see Chapter 2 for more detailed examples), some of the program generator generation-time representations of *pe-values* will indeed have dynamic binding time tags. Glück's "online BTA" strategy [43] avoids this problem by calculating binding time tags only from other binding time tags. Since all such calculations are, by definition, static, and thus performable at program generator generation time, the resultant program generator will contain no residual binding time reductions. The cost, however, is the same as that of offline BTA; binding times calculated using only knowledge of other binding times (and not of specialization-time values) are necessarily conservative and inaccurate. Program generators constructed in this manner will be unable to make certain optimizations because they will incorporate overly general binding time assumptions.

Thus, we see that, to build efficient program generators from true online specializers like TINY, the "outer" specializer `specialize` must handle partially static structures.

Another representation problem has to do with TINY's choice of a specialization-time representation for runtime values, *pe-values*. At program generator generation time, we distinguish four different categories of *pe-values*:

1. Known binding time, known value/expression,
2. Known binding time, unknown value/expression,
3. Unknown binding time, known value/expression, and
4. Unknown binding time, unknown value/expression

Choices (1) and (4) are both easy to implement, since in (1), the *pe-value* is completely static, and can be easily represented, while in (4), the *pe-value* can be represented by `<dynamic>`. Choice (3) is unrealistic, since (at least in the non-partially-static version of TINY) the value/expression is sufficient to allow the binding time to be deduced. The problem, then, is how to represent *pe-values* with known binding times but unknown value/expression fields.

Recall that, in Section 7.2.2, we noted that a *pe-value* is a disjoint union of a value and expression, which has several possible representations. TINY uses a representation which separates the union tag from the value/expression field; this allows us to provide a static tag and a dynamic value/expression. An alternate representation used in some specializers optimizes space usage by using constant expressions of the form `(quote <value>)` to denote static values. Thus, a static *pe-value* is denoted by a `quote` expression, while

a dynamic *pe-value* is denoted by a variable, `if`, `let`, or `call` expression. Unfortunately, this allows us to denote, at program generator generation time, a static *pe-value* with a dynamic value (*i.e.,* `(quote <dynamic>)` but does not allow us to denote a dynamic *pe-value* with an unknown residual code expression, because the dynamic binding time cannot be distinguished from the expression. To handle this encoding, the outer specializer `specialize` would have to be able to denote either negations (*i.e., not* `(quote <value>)`) or disjoint unions (*i.e.,* `<dynamic-symbol>` *or* `(if . <dynamic>)` *or* `(let . <dynamic>)` *or* `(call . <dynamic>)`); such technology is not presently available. Thus, we will use the `(<tag> <value/expression>)` encoding of TINY, which requires only that `specialize` handle partially static structures.

Preserving binding time information

The representational details described in the previous subsection, namely the use of a `(<tag> <value/expression>)` encoding for TINY *pe-values*, and the handling of partially static structures in `specialize`, are sufficient to generate efficient program generators for many programs, including the interpreter fragment of Section 7.2.3. However, without additional mechanisms in `specialize`, some programs will still lead to inefficient program generators. The problem arises when `specialize` builds a residual loop; if the usual strategy of returning `<dynamic>` out of all calls to residual procedures is used, binding time information will be lost at that point. Consider the `append` program:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b))))
```

Suppose that we specialize TINY on this program, with `a` known to be static (but with unknown value) and `b` known to be a particular static value. During program generator generation, TINY will attempt to unfold the `append` procedure repeatedly on its static first argument. Since the value of that argument is unknown, `specialize` will build a specialized version of TINY's unfolding procedure, specialized on the `append` procedure and an environment where `a` and `b` are bound to values with static binding times. This specialized unfolder will contain a recursive call to itself; which, for the program generator to be efficient, must be shown to return a static value. Otherwise, the program generator will contain code for the case where the return value is dynamic, even though it will always

be static. Figure 7.7 shows examples of program generators for `append` obtained with and without return value reasoning in `specialize`. Note that this is not a problem for offline specializers because (an approximation to) the binding time of `append`'s return value is computed prior to specialization; the reduce/residualize decision for `cons` is made by consulting this binding time annotation rather than the return value of the specialized unfolder.

Of course, it is unlikely that anyone would choose to build a program generator for `append`. However, similar recursive procedures appear frequently in realistic programs like interpreters. For example, the procedure `init-names` in the MP interpreter of Figure 7.4 constructs the list of identifiers in the store by recursively traversing a portion of the source program which is static, but with an unknown value, at program generator generation time. This reduces to the same problem as the `append` example above.

Thus, our hypothetical specializer `specialize` must be able to infer information about static portions of values returned by calls to specialized procedures.[10] When we consider more powerful versions of TINY (*c.f.* Section 7.4), we will need corresponding improvements in the information preservation mechanisms of `specialize`.

Example

Thus, we have seen that `specialize` must handle partially static structures and must be able to infer information about static portions of values returned by calls to specialized procedures. FUSE meets both of these requirements, so we can use it to specialize TINY.

Now that we have the tools we need, we can demonstrate the generation of an efficient program generator for the interpreter fragment of Figure 7.3, obtained by specializing the online specializer TINY with the online specializer FUSE.[11] Executing the forms

```
(define names (make-static-peval <dynamic>)
(define values (make-dynamic-peval))
(specialize tiny-program fragment-program (list names values))
```

[10] If TINY were written in a truly tail-recursive style, such as continuation-passing style, `specialize` would not have to reason about return values, but would instead have to reason about static subparts of arguments to continuations with multiple call sites, which is a problem of similar difficulty. See Chapter 5 and [97, 94] for examples.

[11] What's important here is that TINY is online; it doesn't matter whether the specializer used to specialize it is online or offline, as long as it is sufficiently powerful. However, we do believe that the criteria for `specialize` are easier to achieve using online techniques.

$$
\begin{aligned}
PE_{\texttt{(append a b)}}\ env = {} & let\ p = lookup_{\texttt{a}}\ env\ in \\
& (null?\ p) \rightarrow \texttt{(1 2)}, \\
& \quad let\ env' = update_{\texttt{a}}\ env(cdr\ p)\ in \\
& \quad\ let\ p' = PE_{\texttt{(append a b)}}\ env')\ in \\
& \quad\ (p' \in Val) \rightarrow (cons\ (car\ p)\ p'\ ), \\
& \quad\ \ [\![\texttt{(cons}\ (car\ p)\ p'\texttt{)}]\!]
\end{aligned}
$$

Program generator fragment obtained without return value reasoning

$$
\begin{aligned}
PE_{\texttt{(append a b)}}\ env = {} & let\ p = lookup_{\texttt{a}}\ env\ in \\
& (null?\ p) \rightarrow \texttt{(1 2)}, \\
& \quad let\ env' = update_{\texttt{a}}\ env(cdr\ p)\ in \\
& \quad\ (cons\ (car\ p)\ (PE_{\texttt{(append a b)}}\ env'))
\end{aligned}
$$

Program generator fragment obtained with return value reasoning

Figure 7.7: Results of specializing TINY on the `append` program with and without return value reasoning in `specialize`. At program generator generation time, `append`'s first argument is known to be static, but with a dynamic value; its second argument is known to be the static list `'(1 2)`.

$$PE_{\texttt{(cons (car names) (car values))}}\, env = \mathit{let}\ p_1 = \mathit{lookup}_{\texttt{names}}\, env\ \mathit{in}$$
$$p_2 = \mathit{lookup}_{\texttt{values}}\, env\ \mathit{in}$$
$$\mathit{in}\ [\![\texttt{(cons (quote (}\mathit{car}\ p_1\texttt{)) (car }p_2\texttt{))}]\!]$$

Figure 7.8: Fragment of an efficient program generator constructed from TINY

```
;;; executes in an environment where T14 is bound to the pe-value for "names"
;;; (cadr T14) returns the value slot of this pe-value
;;; thus (caadr T14) returns the car of "names"
;;; note that no binding time tags are examined
(list
 '(dynamic ())                                                  ; build a dynamic pe-value
  (list 'code-prim-call                                         ; which is a call
        '(code-identifier cons)                                 ; to the primop cons
        (cons (list 'code-constant (caadr T14))                 ; on a constant (car names)
              '((code-prim-call (code-identifier car)           ; and a call to car
                                ((code-identifier values)))))) ; on identifier values
```

Figure 7.9: The fragment of Figure 7.8, expressed as a Scheme program

returns an efficient program generator which has, "built-in," not only the syntactic dispatch and static environment lookup, but also the binding times of `names` and `values`. An abstract version of a fragment of this program generator is shown in Figure 7.8; the corresponding Scheme code from the actual program generator is shown in Figure 7.9. The efficiency of this program generator is comparable to that of one produced by specializing an offline specializer on the same fragment [15, 12], with the exception of an additional tagging operation to inject the returned residual code fragment into a dynamic *pe-value*.

7.3.3 Examples

In this section, we give several examples of program generators constructed by specializing TINY on inputs with known binding times, and analyze their performance relative to "naive" program generators constructed on inputs with unknown binding times. The text of these program generators is large and fairly uninteresting, so we will instead describe the program generators in terms of their sizes, speeds, and the number of binding time

comparisons they perform.

The tests

We tested our program generator generation method on three programs: the `append` program, a regular expression matcher, and the MP interpreter. We constructed a program generator for each of these programs by using FUSE to specialize TINY on that program, and on the binding time values of its inputs. We ran the resultant program generators on one or more actual inputs, and compared the runtime with that of running TINY on the programs and their inputs directly. The example suites were:

- **append**: The program generator was constructed by specializing TINY on the append program, a static[12] first input, and a dynamic second input. The program generator was executed on two inputs:
  - **append(1)**: first argument = `'()`
  - **append(2)**: first argument = `'(1 2 3 4 5 6)`
- **matcher**: The program generator was constructed by specializing TINY on the regular expression matcher program, a static pattern, and a dynamic input stream. The program generator was executed on one input:
  - **matcher**: pattern = $a(b+c)^*d$
- **interpreter**: The program generator was constructed by specializing TINY on the MP interpreter, a static program and a dynamic input. The program generator was executed on two inputs:
  - **interpreter(1)**: program = comparison program (*c.f.* Page 225)
  - **interpreter(2)**: program = exponentiation program (*c.f.* [12])

7.3.4 Results

Before we continue, we should note that the specialized programs obtained by direct specialization, execution of an efficient program generator, and execution of a naive program

[12] We mean, known to be static, but with value unknown until program generation time.

generator were identical, modulo renaming of identifiers. This renaming arises because TINY uses a side-effecting operation, `gensym`, to create identifiers, and FUSE does not guarantee that the effect of such operations will be identical in the source program (TINY) and the residual program (the program generators). If TINY were purely functional, the specialized programs obtained by all three means would be, by definition, identical.

The program generators produced for the `append` and matcher examples contained no residual binding time tests outside of the cache lookup routine, which must compare binding time tags because it is comparing tagged values.[13] The program generator produced for the MP interpreter does contain binding time checks for operations depending on values in the store (in Figure 7.4, all code depending on the formal parameter `vals`) because it cannot be shown at program generation time that this value is dynamic. If the MP program being interpreted doesn't declare any input variables (*i.e.*, it computes a constant value), then its store will be static even if its input is dynamic. Thus, the generated program generator is willing to make reductions based on known values in the store even though the entire store might not be static.[14] These tests can be eliminated by manually inserting generalization operations, but their elimination does not significantly alter the performance of the program generator.

A comparison of the speed of the specializer and our program generators is shown in Figure 7.10. The speedup ratios, ranging from 3.5 to 20.9,[15] are competitive with those reported by other work on program generator generation [64, 12, 83].

These figures describe benefits due to the use of a program generator, but do not indicate how much of this benefit is due to the use of an *efficient* program generator. To determine this, we constructed naive program generators from TINY by specializing it on

[13] In a polyvariant online specializer, the cache entries for different specializations of the same procedure may have different binding time signatures; thus, the cache lookup code must compare those signatures, which are not available until program generation time (since the cache contents are dynamic at program generator generation time). This tagging problem does not occur in offline specializers, even those with polyvariant BTA, because all polyvariance with respect to binding time signatures has been expressed via duplication at BTA time. When specializing any particular call site, the specializer (program generator) need only consult a cache, all of whose keys have binding time signatures *known* to be equivalent to the signature of the arguments at the call site. Thus, no tag checks are required.

[14] This makes a lot more sense if the specializer handles partially static structures; TINY will only be able to perform such "optimization" reductions when the entire store is static.

[15] This wide variance can be accounted for by noting that only some portion of the program generator's runtime depends on its inputs; for small inputs, the overhead of cache manipulations, etc, which cannot be optimized to the same degree as syntactic dispatch, will dominate. For example, as the static input to the program generator for `append` increases in length, the amount of time spent in the (highly optimized) unfolding procedure increases.

program	time to specialize using TINY	time to specialize using program generator	speedup
append(1)	14	4.0	3.5
append(2)	143	7.3	19.6
matcher	401	71.0	5.6
interpreter(1)	2449	119.0	20.6
interpreter(2)	4346	208.0	20.9

Figure 7.10: Speedups due to program generator generation, for various examples. All times are given in msec, and were obtained under interpreted MIT Scheme 7.2 on a NeXT workstation. The times given are the average over 10 runs, and are elapsed times (no garbage collection took place). The corresponding times for compiled MIT Scheme are 5-35 times faster, with somewhat lower (approx. 30% lower) speedup figures, presumably due to constant folding, inlining, and other partial evaluation optimizations present in the compiler. Timings include only specialization, not pre- or postprocessing.

program	time (naive)	time (efficient)	speedup
append(1)	4.1	4.0	1.0
append(2)	13.3	7.3	1.8
matcher	98.3	70.8	1.4

Comparison of execution times or naive and efficient program generators

program	size (naive)	size (efficient)	size ratio
append	1071	428	2.5
matcher	32983	874	37.7

Comparison of sizes of naive and efficient program generators.

Figure 7.11: Comparison of execution times and sizes of naive and efficient program generators. We were unable to construct a naive program generator for the MP interpreter within a 32MB heap; thus, no data are provided for this case. Times are in msec, while sizes are in conses.

completely (program generator generation time) dynamic arguments, and compared the performance and size of the resultant program generators with the efficient program generators constructed above. The results (Figure 7.11) show that the naive program generators are slower than their efficient counterparts, but are still significantly faster than direct specialization. Of the speedup obtained over direct specialization, 0-46% is traceable to the incorporation of binding time information (and the elimination of binding time reductions). The size ratios are more striking: the naive program generators are 2-37 times larger than the efficient program generators. These numbers are larger than those reported in [12], presumably because Similix factors out primitives into abstract data types (which results in operations like `peval-car` or `car` in the program generator), while FUSE beta-substitutes the entire bodies of TINY's primitives. The naive program generators also took correspondingly longer to generate: specialization of TINY with FUSE took 1.2-5.9 times longer, and code generation up to 81 times longer (due to inefficiencies in the current implementation of the FUSE code generator). Large program generators can cause other problems with the underlying virtual machine; *e.g.,* we were unable to compile the naive program generator for the matcher, which contained a 7000-line procedure, in a 32MB heap.

Thus, we see the benefits of restricting the generality of program generators by providing binding time information at program generator generation time.

7.4 Extensions

The program specializer TINY used in the experiments of Sections 7.3 and 7.3.3 is very simple. In this section, we describe several classes of extensions to TINY, and how they affect the difficulty of producing efficient program generators by specializing TINY. We begin (Section 7.4.1) by adding online generalization, then partially static structures (Section 7.4.2). Section 7.4.3 treats two other mechanisms, induction detection and fixpoint iteration, used in online specializers, while Section 7.4.4 summarizes the difficulties in specializing online specializers, and what is needed to solve them.

7.4.1 Online Generalization

As described in Section 7.2, TINY uses an offline strategy for limiting unfolding and for limiting the number of specializations constructed: procedures are explicitly annotated as unfoldable or specializable, and parameters are explicitly annotated as whether they should

be abstracted to "dynamic" before specialization is performed.

In Chapter 2, we argued that one of the main strengths of online program specializers is their ability to perform *generalization* operations online. Specializers with such a mechanism discard information only when necessary to achieve termination, retaining static values which are common to multiple call sites sharing the same specialization. Thus, it would be desirable if such mechanisms did not have adverse effects on the generation of program generators.

We extended TINY to perform online generalization as follows. Each formal parameter of each user function definition is annotated according to whether it is guaranteed to assume only a finite number of values at specialization time (such annotations can be computed offline, as in [54]). The specializer maintains a stack of active procedure invocations; if it detects a recursive call with identical finite arguments, it builds a specialization on the generalization of the argument vectors of the initial and recursive calls; otherwise, it unfolds the call. For efficiency's sake (*i.e.,* to reduce the size of the stack, and the cost of traversing it at specialization time) we also add an annotation to calls which will always be unfoldable (this can also be computed statically; any nonrecursive procedures, or procedures whose recursion is controlled by finite parameters, can always be unfolded). Because TINY doesn't implement partially static structures, the output of generalization is almost always "dynamic," so there is less to be gained by online generalization than in partially static/higher-order specializers like FUSE; the point here was to determine if this stack mechanism adversely effected the specialization of TINY. Thus, the specializations produced by the version of TINY with online generalization are not appreciably faster than those produced by the version using offline abstraction techniques.

When we specialized the enhanced TINY on the MP interpreter, we still obtained an efficient program generator. Unlike the program generators constructed from the original TINY, this program generator contains specialized unfolding *and* specialized specialization procedures for some of the procedures in the MP interpreter, namely those not annotated as unfoldable. The specialized specialization procedures incorporate all binding time operations on parameters marked as finite (*i.e.,* the program generator performs no tests which will always take only one branch at program generation time) but do contain residual binding time tests for parameters computed via generalization (because the binding times of these parameters are not known until program generation time). This is exactly what we

want: statically determinable (*i.e.,* determinable at program generator generation time) operations have been incorporated into the program generator, while operations which cannot be (accurately) performed until program generation time (such as generalization and operations depending on the outputs of generalization) are performed by the program generator.

The version of TINY with online generalization is slower than the version of TINY with static generalization annotations: in the case of the MP interpreter, specialization took 1.9-2.2 times longer, depending on the program being specialized. This ratio carried over into the program generators; the program generator with online generalization was 1.8-2.1 times slower. The speedup due to the use of a program generator instead of direct specialization remained almost unchanged when online generalization was introduced.

Thus, we do not believe that online generalization is an obstacle to the generation of efficient program generators, with some caveats. The outputs of the generalization routine are, naturally, unknown at program generator generation time, so both the generalizer and operations depending on the output of the generalizer are left residual in the program generator. The key to constructing an efficient program specializer with online generalization lies in determining, at program generator generation time, which parameters are liable to be generalized and which aren't. TINY accomplishes this using static finiteness annotations: anything promised to be finite won't be generalized. Without such a static scheme, the outer specializer `specialize` would have to implement complex mechanisms such as equality constraints to prove that certain parameters to the generalizer would be guaranteed to be equal at program generation time, allowing the incorporation of the parameters' binding times into the program generator. Thus, we see that offline methods are useful even in the case of online specialization, particularly with respect to termination. This stands to reason—we are not arguing that all operations are best performed online, merely that performing some operations (such as binding time and generalization operations) online has benefits and does not adversely affect the efficiency of program generation.

7.4.2 Partially Static Structures

TINY does not implement partially static structures; that is, a pair containing one static value and one dynamic value at specialization time is treated as a completely dynamic value. In some cases, partially static structures can be "teased apart" into static and dynamic components either manually or automatically [83, 34], but this is not always possible. Thus, TINY's lack of partially static structures is a limitation.

```
(define (init-store names values)
  (if (null? names)
      '()
      (cons (cons (car names) (car values))
            (init-store (cdr names) (cdr values)))))
```

Figure 7.12: Code to initialize a store represented as an association list

As we shall see, adding partially static structures to TINY while retaining the ability to produce efficient program generators strains the limits of current specialization technology. We will describe two possible encodings of partially static structures in TINY, and how they affect the specialization of TINY.

Two-tag encoding

Our first encoding scheme retains the structure of the existing TINY encoding, but changes its interpretation. *Pe-values* are still represented as tagged values with the tags `static` and `dynamic`, but the value slot of a static *pe-value* must either be an atomic Scheme value or a Scheme pair containing two *pe-values* (instead of two Scheme values, as before). The encoding of dynamic values is unchanged. Thus, a partially static value is simply a static pair containing a dynamic element. No information about completely static values is maintained, as it is not useful in performing reductions (the code generator will find completely static subtrees and generate `quote` expressions for them, instead of chains of `cons` primitives). The primitive application, cache lookup, and finiteness annotation mechanisms are changed to accommodate this new representation, but, overall, TINY isn't changed much.

This small change to TINY makes the generation of efficient program generators tremendously difficult; to maintain binding time information encoded in this form, the outer specializer, `specialize`, must be able to infer and maintain information about disjoint unions and recursive data types. Consider the function `init-store` (Figure 7.12), which takes a list of names and a list of values and constructs an association list mapping each name to the corresponding values. Assume that we wish to construct a program generator for an interpreter containing this function, where `names` is derived from the interpreter's program input, which is known to be static at program generator generation time, while `values` is

derived from interpreter's data input, which is known to be dynamic at program generator generation time. We would expect the program generator to contain a residual loop to construct the store initialization code, and we would expect that the residual loop would be known to return a list of unknown length, but where each element was known to be a pair whose `car` is known to be static, and whose `cdr` is known to be dynamic. This would allow later operations involving the names in the store to be reduced (*i.e.,* the program generator contains specialized code to perform these reductions), while operations involving the values would be residualized (without a prior examination of their binding times).

Thus, at program generator generation time, `specialize` must be able to represent the following types:[16]

- A completely static *pe-value*; that is, the tag is `static` and if the value is a pair, both the `car` and `cdr` are also completely static *pe-values*.

  ```
  <t1> ::= (static [() | (<t1> . <t1>)])
  ```

- A *pe-value* with tag `static` and whose value is either the empty list or a pair. If the value is a pair, its `car` is a *pe-value* with tag `static` and a value which is a pair of a static *pe-value* with unknown atomic (non-pair) value, and a dynamic *pe-value*, and the recursive type.

  ```
  <t2> ::= (static [() | ((static ((static <atom>) . (dynamic <dynamic>)))
                          . <t2>)])
  ```

- A *pe-value* with tag `dynamic`.

  ```
  <t3> ::= (dynamic <dynamic>)
  ```

The third type is simple, but the first two are rather complex. Indeed, we know of no program specializer capable of inferring (or even representing) such types; even FUSE only

[16] We will use identifiers, parentheses, and periods to denote `specialize`'s representations of Scheme objects, while angle brackets and alternate constructions of the form `[<a> | <b>]` will denote meta-objects of `specialize`. Thus, `<t> ::= [() | ((1 . 2) . <t>)]` denotes a list of unknown length consisting of pairs whose `car` is 1 and whose `cdr` is 2. The special meta-objects `<dynamic>` and `<atom>` denote values not known at the time `specialize` runs; in addition, `<atom>` is constrained to denote only atomic Scheme values.

maintains information about types whose size[17] is known at specialization time, reverting to the type `<dynamic>` for any specialization-time value which might denote arbitrarily large values at runtime. Inferring recursive types is very difficult, because `specialize` must choose when to collapse a chain of disjoint unions into a recursive type, and must be able to compare and generalize such types. This is difficult because the "collapsing" operation can be done in different ways, with different results; such difficulties are standard when analyzing pointer data structures [20, 51]. Existing approaches to this problem in the pointer analysis community have used either k-bounded approximations, which limit the size of nonrecursive type descriptors, or methods based on limiting each program expression to returning a single type descriptor. This latter approach is also used by the partially static binding time analyses of Mogensen (one binding time grammar production per program point in [83]) and Consel (one type descriptor per *cons point* in [22]), and in the monovariant type evaluator of [118].

Such approaches work well for analyzing an existing program, because the identity of the expressions in the program can be used as "anchor points" to perform least upper bounding and build recursions (as is done in [61, 82, 23]). In the case of an online specializer, which must infer the type of residual code as it is being constructed, we lose this ability to use the identity of code expressions; during the iterative type analysis process, several residual code expressions may correspond to a single program point in the source program. To achieve termination (*i.e.,* to avoid infinite disjoint unions) some of these code expressions (and their types) must be collapsed together, but we can't just collapse together all instances of a source program point as this would yield a purely monovariant specialization. The problem, then, lies in deciding when and how to collapse, or generalize.

Consel's polyvariant partially static BTA [22] operates by keeping all nonrecursive invocations of a procedure distinct, but collapsing recursive call sites together with initial call sites; this works fine for BTA, but will not work for online specialization, as it precludes unfolding of recursive procedures. Aiken and Murphy's type analyzer [3, 84] appears to use a similar method. Mogensen's higher-order partially static polyvariant binding time analysis [83] is for a typed language, and uses (user- or inferencer-provided) declarations when deciding what recursive types to construct.

One promising approach is the two-stage partial evaluation framework of Katz [70], in

[17] By "size," we mean "number of `cons` cells contained." Even numeric types can become arbitrarily large at runtime, but they are still scalars; there is no need for recursive type descriptors to describe bignums.

which a polyvariant analysis phase computes type information which is then utilized by a separate code generation phase. This offers the possibility that the polyvariant analysis phase could make use of the identity of source program expressions (possibly tagged with some form of instance counter) for building recursive types, without being confused by the construction of multiple residual instances of single program points due to fixpoint iteration and unfolding.

Thus, given the current state of the art, we are unable to construct the first two types listed above at program generator generation time. The consequences of this are disastrous, as we are able to accurately represent only dynamic *pe-values*, and static *pe-values* of known size. All static or partially static *pe-values* of unknown (at program generator generation time) size end up being represented as `<dynamic>`, which contains no binding time information whatsoever. In the case of the interpreter fragment of Figure 7.12, all binding time information about the parameter `names` and about the value returned by `init-store` is lost. Thus, the resultant program generator will not know that the list of names is static, or that store lookup can be performed statically—indeed, even syntactic dispatch will perform needless binding time comparisons. The program generator will be almost as slow (and large) as a naively generated program generator.

This bodes ill for the self-application of specializers like FUSE, which uses a *symbolic value* encoding similar to the two-tag encoding described in this section. Successful specialization of such specializers appears to require either a change of encoding, or new and more powerful specialization techniques.

Three-tag encoding

In this section, we consider a different encoding of partially static structures, this time using three tag values. The tags `static` and `dynamic` are interpreted as before: `static` denotes a *completely* static value, and is followed by a Scheme value, while `dynamic` denotes a *completely* dynamic value , and is followed by a residual code expression. We add a new tag, `static-pair`, which is followed by a pair of *pe-values*, rather than Scheme values, denoting a pair whose subcomponents are denoted by the corresponding *pe-values*.[18] Thus, `(static-pair ((static 1) . (dynamic (foo x))))` denotes a pair whose `car` is `1` and whose `cdr` is unknown, but can be constructed by the code `(foo x)`.

[18] Of course, a partially static value must also contain the appropriate residual code fragment for constructing the value at runtime. Since this attribute is immaterial to our discussion, we will ignore it.

Compared with the two-tag encoding of Section 7.4.2, this encoding requires slightly more mechanism in the program specializer (TINY) because the `cons` primitive must consult the binding times of its inputs to decide how to tag its output (instead of just always tagging it `static`). Similarly, the code for comparing argument vectors in the cache (and, in the case of online generalization, the stack) must be able to traverse static and partially static pair structures in parallel. However, we will see that this added cost allows more efficient program generators to be generated using existing technology.

Consider the `init-store` code of Figure 7.12. At program generator generation time, `specialize` must be able to represent the following types:

- A completely static *pe-value*; that is, the tag is `static`.

```
<t1> ::= (static <dynamic>)
```

- A *pe-value* which is either the empty list or a partially static pair whose `car` is a partially static pair with static `car` and dynamic `cdr`, and whose `cdr` is the recursive type.

```
<t2> ::= [(static ()) |
          (static-pair ((static <dynamic>) . (dynamic <dynamic>))
                       . <t2>)]
```

- A completely dynamic *pe-value*; that is, the tag is `dynamic`.

```
<t3> ::= (dynamic <dynamic>)
```

Both the first and third types are easy for `specialize` to represent, as it knows the size of all static parts (*i.e.,* both are tuples of length 2). The second type still requires recursive type inference, which is beyond the current state of the art.

Thus, when TINY is specialized on an interpreter containing a call to `init-store`, the resultant program generator will not contain any binding time comparisons for `names` or `values` (or for any syntactic dispatch operations) but will contain needless binding time comparisons in the store lookup routine, because `specialize` lost the information about the structure of the store (*i.e.,* an association list with static `car`s). All binding time operations for completely static or completely dynamic operations are reduced at program generator

generation time, but such operations on partially static structures (or their components) are performed online in the program generator. This is significantly better than the results obtained with the two-tag encoding, which couldn't even optimize out operations on completely static structures, but not as good as could be obtained with a more powerful version of `specialize`. Specializing this version of TINY on the MP interpreter using FUSE yielded speedup figures of 11.4-12.0, depending on the MP program being specialized. Because binding time operations on partially static structures are not performed at program generator generation time, this program generator still performs unnecessary binding time manipulations on the store; removing these manipulations would achieve better speedups.

This is an instance of a common phenomenon in partial evaluation, namely the extreme sensitivity of the specializer to changes in the representations used by the problem being specialized. Indeed, the use of a less efficient representation (such as the three-tag encoding here) can often lead to more efficient specializations if the extra work (in this case, having TINY's `cons` primitive consult the binding time tags of its arguments to determine the tag for its result) is performable at specialization time.

7.4.3 Other Online Mechanisms

In addition to binding time tests in primitives, and generalization, some specializers perform other operations online, such as induction detection [115] and fixpoint iteration (*c.f.* Chapter 4 and [116, 97]). These operations, not present in TINY but present in some versions of FUSE, significantly complicate the task of producing efficient program generators. In this section, we will briefly describe these mechanisms, and explain why present specialization technology cannot handle them.

Induction Detection

Some online specializers such as FUSE and Mixtus [99, 100] make unfold/specialize decisions automatically during specialization. This is usually a two-step process: (1) a recursive call is detected (using a specialization-time call stack), then (2) the specializer decides whether the recursive call poses a risk of nontermination. If there is no risk, the call is unfolded; otherwise, its arguments are generalized with those of the prior call, and a specialization is constructed on the resulting argument vector.

A variety of mechanisms are used for (2); for example, some versions of FUSE generalize only if there is an an intervening dynamic conditional between the initial and recursive calls,

thus executing all non-speculative loops (even infinite ones) at specialization time. Another common mechanism is induction detection, which works as follows. The specializer assigns a well-founded partial ordering to all elements of the domain of argument vectors, and unfolds a recursive call only when the recursive call's argument vector is strictly smaller (in the partial ordering) than the prior call's argument vector. This detects and unfolds static induction, such as `cdr`-ing down a list of known length, or counting down from some number to zero (this only works if a "natural number" type is provided; otherwise, we wouldn't know that the induction is finite at specialization time). Of course, it misses many cases where the iteration space, though bounded, doesn't map monotonically to the partial ordering (consider a list-valued argument which shrinks by two pairs, grows by one, shrinks by two, etc). However, without help from the programmer in the form of annotations, we cannot hope to detect all such inductions (thus the use of finiteness annotations in some versions of FUSE).

The use of such an induction technique complicates the task of program generator generation because of the need to decide the domain comparisons at program generator generation time. Consider the procedures `mp-command`, `mp-begin`, and `mp-exp` in the interpreter of Figure 7.4; the syntactic arguments (`com` and `exp`) always strictly decrease on recursive calls, so the specializer will unfold such calls. However, at program generator generation time, only the static nature of these arguments is known; the values are not. When a simple outer specializer `specialize` evaluates the unfold/residualize decision procedure on `(mp-exp (car rest) names value)`, it knows that `(car rest)` is bound to a value with tag `static`, but it doesn't know that the value is strictly smaller than the prior value, `exp`; it just sees `(static <dynamic>)` as the value for both `pe-values`. This means that the comparison is not decidable at program generator generation time, so specialized procedures for both specialization and unfolding are constructed. This increases the size of the program generator, but does not significantly affect its speed. Avoiding this case requires that `specialize` perform inductive reasoning about dynamic values; it would have to notice that the dynamic value in the recursive *pe-value* is derived from the dynamic value in the initial *pe-value* using a string of `car` and `cdr` operations, which means the new value is smaller. It is possible that offline induction analyses such as that of Sestoft [103] could be adapted for this purpose.

However, using this information to make TINY's unfold/specialize decisions at program generator generation time could be difficult; just because `specialize` knows that one list

is shorter than another doesn't mean that it can fully evaluate TINY's decision procedure. Because the absolute lengths of the lists are unknown (only relative information is available), the body of the comparison loop (and the `null?` tests contained in the body) cannot be unfolded. Instead, `specialize` must perform theorem proving, using the relative length information to show that, no matter how many iterations the comparison loop performs, its result will always be the same (this can be done by propagating information from the tests of dynamic conditionals (`null?` tests) into the arms, which is not performed by most existing specializers). A simpler solution relies on explicit reflection [106] in the form of an upcall (*i.e.,* instead of explicitly implementing the "shorter" predicate on lists in TINY, make it a primitive in `specialize`, which simply checks to see if one of the lists is derived from the other). However, this would be a less versatile technique; we believe that the results of any reasoning in a specializer such as `specialize` should be usable for improving *any* program, not just special programs containing upcalls.

Some cases are even worse. When `specialize` encounters the decision procedure on the recursive call to `mp-while` in `mp-while`, it will build specialized unfolding and specialization procedures. The specialization procedure will be invoked on the generalization of the current and prior arguments to `mp-while`. Though we can see that the syntactic argument, `com`, is identical in both calls, `specialize` cannot, and thus builds a specialization routine which doesn't know that the binding time of `com` is static, leading to needless binding time comparisons not only in the specialization routine for `mp-while`, but also in the unfolding/specialization routines for any procedures it invokes, such as `mp-command` and `mp-exp`. In short, all knowledge of the static binding time of syntactic arguments is lost, and the program generator is as inefficient (slow and large) as a naively generated one. Avoiding this requires that `specialize` infer and maintain information about the equality of dynamic values, so that when TINY does an `eq?` or `equal?` check on the (dynamic) Scheme value fields of the two static *pe-values*, `specialize` will return true instead of residualizing the decision. Existing specializers do not perform such reasoning, though there is hope that they eventually will, because any specializer hoping to perform well on imperative programs with pointer structures must have such an equality analysis to handle aliasing.

For now, the construction of efficient program generators from specializers using online generalization mechanisms requires that we have some static means of identifying those arguments which cannot possibly be raised to dynamic via generalization, such as those arguments marked with "finite" annotations in TINY. Offline specializers, which by their

very nature are prohibited from using online induction analyses, satisfy this restriction trivially, but are of course unable to benefit from such analyses.

Fixpoint Iteration

Another mechanism by which online specializers gain accuracy advantages over their offline counterparts is via fixpoint iteration analyses such as those described in Chapter 4. For example, FUSE can determine that any number of functional updates to a store represented as an association list will preserve the "shape" of the store—that is, the `car`s will remain unchanged. This is important because it avoids unnecessary searching in programs constructed by specializing interpreters.

Unfortunately, such analyses encounter problems similar to those faced by specializers with induction detection and online generalization. The problem is that the inner specializer (TINY) decides the equality of two static specialization-time values (for example, the values of parallel keys in two different association lists) by executing the Scheme procedure `equal?`, which cannot be evaluated at program generator generation time, as only the binding times (and not the values) are available. In fact, the two keys will be equal no matter what values are provided at program generation time; their equality is a property of the interpreter, independent of the program on which it is specialized. Making TINY's mechanism work would require that the outer specializer keep track of equality relations between dynamic values, so that the equality tests used for generalization and termination of fixpoint iteration would be decidable at program generator generation time.

This might well be a fruitful area for future research even in the domain of offline specializers, because the static reasoning needed to prove that the shape of a store doesn't change, given only the source text of the interpreter (but not of the program being interpreted) could just as easily be performed at BTA time as at program generator generation time. If such an analysis could be provided, then the fixpoint iterations themselves might become unnecessary—the analysis could prove that the names in the store would remain unchanged across iterations instead of having to rediscover this fact for each different set of names (Of course, if we wish to preserve as much information as possible about the *values* in the store, online generalization is still necessary, because the behavior of the values is determined by the program text, not just the interpreter text. Similarly, an interpreter for a language like BASIC, where the store can potentially grow during a loop due to automatic initialization of undeclared identifiers, requires online methods because the equality of store shapes on

- Basic online PE
  - partially static structures in `specialize`
  - return value computations in `specialize`
  - explicit tag values in TINY or disjoint union types in `specialize`
- Online Generalization
  - static indication of "ungeneralizable" values in TINY (*i.e.,* finiteness analysis) or equality reasoning in `specialize`
- Partially Static Structures
  - recursive types in `specialize`
- Induction Detection
  - size reasoning in `specialize`
  - propagation of information from tests of dynamic conditionals into arms in `specialize` or primitives for size predicates used in TINY.
- Fixpoint Iteration
  - equality reasoning in `specialize`

Figure 7.13: Online features and mechanisms for specializing them

recursive calls is not provable given the interpreter text alone).

7.4.4 Summary

Figure 7.13 summarizes our discussion of the features of online specializers and the mechanisms needed to produce efficient program generators from specializers with such features. Each feature is listed, along with the necessary mechanisms. Together, FUSE and TINY meet these constraints for the first two features, online specialization and online generalization. By choosing appropriate of representations, we achieved some success for partially static structures. Full efficiency with partially static structures, induction detection, or fixpoint iteration will require new specialization technology.

We should also note that Figure 7.13 should probably include a line for side effects. Many online mechanisms can be implemented far more efficiently using side effects (FUSE

makes heavy use of side effects, to data structures as well as to variables), but if side effects are used, then `specialize` must be prepared to reason about them. Existing techniques, which basically residualize all side-effecting or side-effect-detecting computations, will not be sufficient if binding times are represented using side-effectable data structures (contrast this with the offline case, where binding time information is retained no matter how the specializer handles side effects).

7.5 Related Work

This section describes related work on program specialization, with an emphasis on techniques for constructing efficient program generators.

7.5.1 Offline Specialization

Offline specialization and Binding Time Analysis were developed specifically to solve the problem of generating efficient program generators. The first offline specializer, MIX [64, 65] did not handle partially static structures and used explicit unfolding annotations, but was efficiently self-applicable. Subsequent research has produced increasingly powerful offline specializers, which handle partially static structures [83, 22], higher-order functions [83, 12, 23, 45], global side effects [12], and issues of code duplication and termination [103, 12].

The accuracy of offline specializers has been improved through the development of more accurate binding time analyses, such as the polyvariant BTA [22, 83], and *facet analysis* [29], which allows BTA to make use of known properties of unknown values. Other accuracy improvements have been achieved via program transformation, both manually [12] and automatically [83, 34, 26, 55].

In all of the above, efficiency was realized by self-application of the specializer. The usefulness of explicit binding time computations in realizing self-application is described in [15, 12, 62]. Even in the offline world, efficiency methods other than self-application have been used, including the factoring out of computations depending on binding time information (as opposed to only factoring out the binding time computations themselves) [25] and handwriting a program generator generator [56].

Work continues on all of these fronts; however, we do not believe that any current offline specializer is sufficiently powerful to produce an efficient program generator from TINY.

7.5.2 Online Specialization

The earliest program specializers [67, 7, 49] used online methods. None were suitable for program generator generation, though handwritten program generator generators such as REDCOMPILE [49] were used. Subsequent work on online specialization has focused primarily on accuracy [101, 46, 115, 96, 97, 93, 70, 100, 111] and applications [8, 10, 6, 117] rather than on efficiency.

A notable exception to this is the work of Glück, whose online specializer, V-Mix [43], has been used to generate efficient program generators via self-application. Glück's work makes the same observation as our Section 7.3.1, namely, that the problem of excessive generality in program generators can be solved without static binding time annotations by encoding the binding times in the arguments passed to the inner specializer. His formulation of self-application (using the "metasystem transition" formalism of Turchin [111]) is very similar to the formulation we gave in Choice 2 on page 228.

Despite these similarities, however, V-Mix and this work differ appreciably in the encoding and preservation of binding time information. In particular, V-Mix uses a *configuration analysis* (described as a "BTA at specialization time") to make reduce/residualize decisions and to compute static/dynamic *approximations* of function results at specialization time (Bondorf outlines a similar approach in [12], p. 34). We would expect such a system to be self-applicable, since, just as in the offline case, all reduce/residualize decisions are made by a process that refers only to the program text and statically available information (binding times), so there is no danger of losing this information at program generator generation time. Unfortunately, such methods share many of the drawbacks of purely offline methods. For example, binding time approximations to function results computed using only binding time approximations to parameters are overly general—many functions in a program will be given a dynamic return approximation when they might actually return a static value when unfolded at program generation time. Indeed, V-Mix's inability to compute an "unknown" binding time approximation means that all binding time operations will be resolved at program generator generation time, yielding a program generator with *no* online binding time operations even when such operations are necessary to achieve an accurate result.

Indeed, it would appear that the results of configuration analysis could be duplicated by a sufficiently accurate polyvariant binding time analysis. The primary benefit of online specialization in V-Mix appears to be the simplicity with which it achieves polyvariance with respect to binding times, not the accuracy of the (essentially offline) program generators it

produces. In all fairness, we should note that Glück's work dealt primarily with multiple self-application, in which several levels of specialization were performed, yielding a curried residual program in which each of the first $n-1$ arguments produced a new residual program which was then applied to the next argument (application to the nth argument computed the final result). For this application, the "online BTA" approach worked well, with far less memory consumption than the fixpoint computations used by the outer specializer `specialize` in our examples.

7.6 Future Work

In this section, we describe several frontiers for future work in the generation of program generators from online specializers.

7.6.1 Self Application

Although we have demonstrated the specialization of a nontrivial specializer into an efficient program generator, we have not demonstrated self-application; the two specializers (the specializer, `specialize`, and the specializee, TINY) were not the same. Self-application is important if we wish to speed up the process of program generator generation via specialization (*i.e.,* if we specialize the specializer specializing itself, producing a program generator generator, often called a "compiler compiler" in the literature [40, 39, 64]), or if we wish to perform multiple self-application [43] to achieve several levels of currying. It appears as though self-application is achievable only with specializers at particular levels of complexity, where the specializer is simultaneously sufficiently powerful to specialize itself, while being sufficiently simple to be specialized by itself. In the offline paradigm, where the specializer itself performs fairly simple computations directed by static annotations, self-application can be achieved at a relatively low level of complexity.

Unfortunately, this does not appear to be the case for online specializers. Specializing the rather simple specializer TINY required not only partially static structures, but also fixpoint iteration.[19] Specializing a partial evaluator which implements partially static structures require the inference of recursive data types, while fixpoint iteration appears to require inductive reasoning. That is, we have yet to implement mechanisms of sufficient power to

[19]This stands to reason because most abstract interpretation-based binding time analyses also require fixpoint computations.

specialize all of the information preservation mechanisms in FUSE, let alone the mechanisms necessary to specialize these as-yet-unwritten mechanisms. More research is required to locate the level of functionality at which closure is achieved, and to determine how to implement such functionality efficiently.

7.6.2 Encoding Issues

Online specializers necessarily incur an overhead due to the need to represent both static values (all Scheme datatypes) as well as dynamic values using a single universal datatype. Offline specializers for untyped languages can avoid this need for encoding because they have no need to encode dynamic values, and can thus inherit the representation of static values from the underlying virtual machine. Launchbury's lazy encoding technique for typed languages [77] reduces encoding overhead for completely static values, but appears to be of less use for partially static values, because, under online methods, the entire spine from the root of a partially static structure to each of its dynamic leaves must be fully encoded at the time the structure is created (how else could dynamic data be distinguished from static values?). This is not a problem in offline systems like Launchbury's because there is no need to encode dynamic values—the specializer determines dynamic-ness from binding time annotations, not from values.

This encoding has not been a problem to date; specializers such as FUSE, though slower than their offline counterparts, operate at acceptable speeds. Users have been willing to pay the time cost in exchange for the improved accuracy. However, program generator generation brings three encoding problems to light:

First, the outer specializer is forced to specialize all of the inner specializer's encoding and decoding operations (indeed, this is how binding times are deduced). This has large costs in both space and time due to the quadratic explosion in the size of the representations of values; *i.e.,* if a specializer executes k instructions per instruction in the program being specialized, then program generator generation takes time k^2. Glück [43] also notes such growth. For small k, as in offline specializers, this is not a problem, but for larger k (compared with offline specializers, online specializers may execute 5-50 times as many instructions per simulated instruction) this growth becomes unacceptable (*i.e.,* if specialization is 5 times slower, program generator generation may be 25 times slower). For example, specializing TINY on the MP interpreter with FUSE required 7 minutes and a 32 megabyte heap.

Second, the generated program generator often encodes values unnecessarily; *i.e.,* the program generator inherits the encoding used by the specializer it was generated from, and encodes (tags) values with binding time information even when those binding times are never examined. Of course, any value which might reach a binding time test must be tagged, but current *arity raising* [90] techniques are insufficiently powerful to remove all "dead" tags. In particular, current arity raisers eliminate only static portions of *parameter* values, without removing static portions of *returned* values. CPS-converting [108] programs will take care of the problem of returned values, but may require a fairly sophisticated dead-code analysis in addition to any complexities added by the higher-order nature of CPS code. Worse yet, if the specializer is written to return tagged values at top level (*i.e.,* FUSE returns a residual program containing, among other things, encoded values), then the program generator must do the same, reducing the number of "dead" tags. It is likely that tag optimization techniques for dynamically typed languages [53, 86] could provide significant improvements here, not only for the specialization of specializers, but the specialization of interpreters for dynamically typed languages as well.

Finally, as we saw in Sections 7.2.2 and 7.3.2, the need to represent values with known binding times and unknown values at program generator generation times constrains the encoding scheme of the inner specializer. In particular, the need to store the binding time and value in separate tuple slots (as opposed to using a distinguished value such as `quote` to indicate static values), enlarges the encoding of values, leading to higher space consumption both at specialization time and at program generation time.

7.6.3 Accurate BTA

Another potential direction for research in program generator generation is the use of binding time analysis techniques which do not force the specializer into overly general behavior. That is, when two abstract "static" values are generalized, the result should be "unknown binding time" rather than "dynamic." Only decisions involving expressions with "static" and "dynamic" binding time annotations would be performed at BTA time; decisions involving expressions annotated as "unknown binding time" would be delayed until specialization time, as in online specialization.[20] Consel describes such a binding time lattice in [22] but

[20]The difference between this approach and traditional offline specialization lies in the fact that, in offline systems, *all* binding times must be computed at BTA time, and *all* reduce/residualize decisions must be completely dictated by the results of the BTA. Thus, a system with an "accurate BTA" would be an

expresses concerns over the pollution of entire expressions due to a single subexpression having an unknown binding time. Bondorf's Treemix [12] uses such an analysis, but its effectiveness is not described.

The motivation behind such methods is twofold. First, it could eliminate the need for encoding values which only reach expressions with known binding times, such as the program in the interpreter example or the pattern in the regular expression example, both of which are known to be static. Second, program generator generation would be simplified: all program generator generation time knowledge of binding times could be provided through the binding time annotations, eliminating the need for the more complex mechanisms of Section 7.3.2.

However, this approach does have some difficulties. For a specializer like FUSE, which can represent typed unknown values, the choice of binding time domain may be difficult; it may be possible to adapt the *facet analysis* of [29] for this purpose. A highly polyvariant analysis would be required; otherwise, the binding times of several variants would be collapsed into one, forcing greater numbers of binding-time-related reductions to be delayed until program generation time. The use of a binding time analysis for online specialization is also complicated by the need to approximate the specializer's termination mechanism at BTA time. Unlike offline specializers, which rely on binding time annotations to perform *abstraction* of arguments (*i.e.,* lift specialization-time-infinite static values such as "counters" to dynamic), online specializers perform *generalization* of pairs of argument values at specialization time. Since the values are unavailable at BTA time, it is not possible to determine if the generalization of two static values will be static, meaning that, under a naive BTA, all static arguments to recursive procedures would be raised to the "unknown binding time" value, losing the two benefits described in the previous paragraph. Finally, there is some question as to the benefits to be gained, since online specializers spend a higher proportion of their effort on operations which cannot be optimized given knowledge of binding times, such as recursion detection, generalization, and other information preservation mechanisms.

(optimized) *online* specializer.

7.6.4 Inefficient Program Generators

The motivation for the use of binding time information at program generator generation time is to increase the efficiency of the resultant program generator by avoiding unnecessary reductions at program generation time. Inefficient program generators, which take advantage of the static values available at program generator generation time (*e.g.,* perform syntactic dispatch and environment lookup operations at program generator generation time) but do not make use of binding time information, are both larger and slower than their efficient counterparts. However, preliminary experiments conducted by the author suggest that the loss in speed may not be particularly large; even *inefficient* program generators are significantly faster than general specializers when it comes to program generation. This suggests that if we could control the size of inefficient program generators, we could construct acceptably efficient program generators without the difficulties inherent in making use of binding time information.

7.7 Summary

We have shown that, given a careful encoding of specialization-time values and a sufficiently powerful specializer, we can construct an efficient program generator from a simple yet nontrivial online program specializer. We believe this to be the first published instance of efficient program generator generator from an online program specializer without the use of binding time approximation techniques. This result is significant because it allows for the automatic construction of program generators which make online reduce/residualize decision, enabling, for example, optimizing "compilers." It is also a demonstration of the power of online specialization techniques, since the information preservation mechanisms used to achieve efficient program generation, are, at present, implemented only in an online specializer, FUSE. Unlike binding time approximations, which address only the specialization of specializers, the information preservation techniques used here can improve the specialization of *many* programs, not just the specializer TINY. Finally, our result may be of interest to the logic programming community, where, in contrast to the functional programming community, most program specializers use online methods [38, 100, 74].

Nonetheless, this result is unlikely to lead to the widespread proliferation of online-specializer-based program generators. The most obvious reason is that, although we can

successfully specialize small specializers such as TINY, we have not developed methods sufficiently powerful to specialize state-of-the-art specializers such as FUSE (*c.f.* Section 7.4). This forces the user into a choice between efficiency and accuracy of specialization; given that the primary motivation for using online techniques is accuracy, we expect that most users would prefer the slower, more powerful, and as-yet-unspecializable systems. We believe that the future of program specialization lies in a mixture of online and offline approaches, in which the additional costs of online specialization are paid only when necessary. We leave this to future research.

Chapter 8

Conclusion

In this chapter, we summarize the ideas described in Chapters 2-7, and describe some remaining open problems in partial evaluation.

8.1 Summary of the Dissertation

We believe that this dissertation makes four contributions to the state of the art of online partial evaluation:

- **Return Value Analysis** (Chapter 4): In addition to caching specializations, cache approximations to their return values, and use these approximations to perform more reductions. The approximations are computed using an iterative method for finding least fixed points.
- **Parameter Value Analysis** (Chapter 5): Instead of specializing closures on completely unknown argument vectors, use control flow analysis to bound the set of call sites, and specialize on arguments just general enough to cover those call sites.
- **Re-use Analysis** (Chapter 6): Keep track of what information was used in performing reductions, and use this to approximate the set of argument values for which a specialization can be safely and optimally reused.
- **Online Program Generators** (Chapter 7): Using a careful encoding of specialization-time values and return value analysis, specialize a nontrivial online program specializer, yielding an efficient program generator without any unnecessary

binding time checks.

These contributions attack several open problems in partial evaluation. The first two analyses improve the accuracy of specialization, producing faster residual programs with less need for manual binding time or staging transformations on the source. Re-use analysis improves both the size of residual programs and the speed of specialization. Online program generators show that the "specializer specialization" technique popularized in the offline community can also be applied to online systems.

Equally important, in our view, is the realization that online techniques can and do provide real benefits. The goal of this work is not so much to "sell" particular algorithms, but rather to motivate, both by analysis and by example, the idea that *making at least some reduce/residualize decisions at specialization time is worthwhile.* Online specialization is *not* merely a "naive" implementation technology, abandoned in the quest for self-application, but rather a class of algorithms worthy of study in its own right. This is not to denigrate offline methods; indeed, it is our belief that the future of partial evaluation lies in a combination of online and offline techniques, each used where it is most appropriate, where that appropriateness may vary by application. It is our sincere hope that, while the debate over appropriateness may continue, any debate over legitimacy has been laid to rest.

8.2 Future Work

In this work, we have made progress on two fronts: improving the *strength* of the program specializer, allowing it to produce faster specialized programs, and improving the *efficiency* of the partial evaluator, allowing it to produce specialized programs more quickly. We believe there is still much to be done in both of these areas, but also in the areas of *automation*, *integration*, and *applications*. This section describes our impressions of open problems and potential solutions in each of these areas. For another treatment of some these issues, we recommend [28].

8.2.1 Strength

By *strength*, we mean degree of optimization, or how much the program specializer is able to speed up the input program. As we have mentioned before, we are concerned not only with how well the specializer can improve a particular program, but with how well it can improve different implementations of the same algorithm. We believe that current specialization

technology, including that of FUSE, is still not strong enough to handle arbitrary end-user code, even in purely functional languages.[1] Program specializers are still very sensitive to both (1) representation decisions, and (2) staging decisions.

Representation

Current partial evaluators operate only on representations; that is, they improve programs by performing semantics-preserving transformations on the primitives used to implement abstract data types. This ability (and need) to "look inside" black-box data abstractions is at once a benefit and a curse. Partial evaluation derives most of its optimization capability from removing procedural and data abstraction barriers, effectively "customizing" generic operations on abstract types. At the same time, however, this focus on representation can lead to difficulties.

First, the partial evaluator can only make use of those properties of the abstract data type which it can discover from the implementation. Given that partial evaluators have imperfect type systems and reasoning capabilities, this makes them very dependent on the user's data representation choices. We saw one example of this in Section 7.4.2, where a change of representation for binding times allowed the partial evaluator to do a better job. As another example, consider an implementation of sets as unsorted lists without duplicates. If the implementation of union conses any new elements onto the front of the list (a constant-time operation), the usual generalization strategy on lists will lose information: the union operation on $\{a, b\}$ and $\{\top_{Val}\}$ yields $(\top_{Val}\ \top_{Val}\ .\ \top_{Val})$.[2] If the implementation instead appends any new elements onto the end of the list, then the result is `(a b .` $\top_{Val}$`)`, which is more useful.

Conversely, the implementation is often overconstrained relative to its specification; that is, some artifacts of the implementation may not be relied upon by other portions of the implementation. Unfortunately, since a typical partial evaluator operates in a compositional manner, preserving the semantics of each expression as it is specialized, it cannot relax any of the implementation constraints. For example, if sets are maintained as unsorted lists, then the sets represented by `(a b c)` and `(c b a)` are equal, and are freely substitutable;

[1] Another aspect of strength relates to the programming language constructs handled by the program specializer; *e.g.,* side effects, error handling, etc. Existing technology can provide *correct* program specialization in the face of such constructs, but usually at the cost of leaving the troublesome constructs residual. Thus, once again, the problem boils down to doing a good job of optimization.

[2] We are assuming a type system without disjoint unions.

e.g., we needn't build two different specializations of a procedure on these two values. Unfortunately, the partial evaluator (1) doesn't know that no operation on the set type can distinguish these two representations, and (2) even if it could derive that, it wouldn't be assured that some user code might not break the abstraction (*i.e.,* if the user takes the `car` of the set representation, he *will* be able to tell the difference. If sets can be returned to the user at top level (who cannot be kept from viewing them as pairs), the partial evaluator must treat the two equivalent representations separately, since their meanings as pairs (if not their meaning as sets) *are* different.

Some partial evaluators have limited support for abstract data types. Schism [21] allows the definition of ML-like types in Scheme, allowing a minor amount of representation independence (*e.g.,* the user needn't know if records are implemented as lists or vectors) but no higher degree of insulation (*i.e.,* if Scheme supported record types, we'd get the same result). Similix [14] allows the user to hide the internals of an operator from the partial evaluator, executing the operator only if its inputs are completely known, and leaving it residual otherwise. This is useful for avoiding code size explosion, but unfortunately prevents any optimization that might take advantage of partial inputs. Thus, there is still work to be done in this area.

We believe that better type systems and type inference can help solve these problems, especially the first case above, but that a general solution will require some help from the programmer and the programming language. Simply hiding implementation details completely and forcing the specializer to operate solely on specifications is insufficient; *e.g.,* it would allow for algebraic optimizations on sets, but would not allow the construction of specialized versions of set operators that could not be completely executed given the specifications. If, in addition, the specializer had some description of the degree to which an implementation of an abstract data type depended upon its representations, it might attempt to get the best of both worlds.

Staging

Another strength issue lies in the sensitivity of partial evaluators to staging [66] decisions made in the source program. We give a few example here.

Most existing specializers miss optimizations when static information must be propagated between loops, because their type systems only describe structured values (pairs and closures) whose size is known at specialization time. This means that, in programs where

one loop produces a list (or stream) of values, which is then consumed by another loop traversing the list (or stream), the "consumer" loop will not be optimized unless the induction variable of the "producer" loop is known and finite.[3] For example, unless we can describe that a function `foo` returns a list of integers, we will not be able to optimize invocations of `+` in a function `bar` that sums the elements of a list. These lost reductions may amount to more than optimization of scalar primitives; consider the problem of partially static binding times in Section 7.4.2. In this case, the problem could be at least partially solved by adding the ability to describe (potentially infinite) recursive structures at some level of detail finer than $\top_{Val}$ or $\top_{List}$.

Other staging problems arise with arithmetic operations. If `a` and `b` are known, and `c` is not, then

```
(+ (+ a b) c)
```

will specialize well, but if `a` and `c` are known, with `b` unknown, then

```
(+ a (+ b c))
```

specializes better. A partial evaluator with knowledge of the associative law for addition might obtain this result automatically, but similar results may hold for user operations and data structures. For example, consider replacing `+` by a user-defined set union operation, where the sets are maintained as sorted lists. Similar issues arise when curried higher-order functions are used. If the known arguments happen to be supplied to the outermost application, all is fine. Otherwise, the outer application will be residualized, and (under all existing higher-order specialization techniques) any inner applications will not be as fully optimized as they otherwise might be.

Still other examples are more application-specific, and more difficult to address by automatic means. For example, in [24], a naive pattern matcher was manually rewritten to explicitly maintain (essentially redundant) information so that, when specialized, the resulting program would run faster.[4] An ideal partial evaluator might have been able to deduce

[3] Turchin's *supercompiler* [111, 112] can optimize the consumer loop even if the producer loop's iteration count is unknown, by combining the two loops into a single residual loop. Unfortunately, this can be performed only when the producer loop is truly recursive, and when the iteration counts of the two loops can be shown to differ only by some small constant. Thus, Turchin's solution, though highly effective in some cases, is not as general as having recursive type descriptions.

[4] This example is too large to describe in detail here; please refer to [24] for listings of the original and rewritten matchers.

the static information from history of the computation, without the need to make it explicit in the input program.

Thus, we believe that type system and type inference mechanisms, such as recursive types, the use of predicate information in arms of irreducible conditionals, constraints between types, will be useful, as will transformation techniques such as driving [111] and currying/uncurrying. Still, we doubt that such issues can be addressed adequately by automatic means.

8.2.2 Efficiency

We see several opportunities for improving the *efficiency* of online program specializers, that is, the speed with which the specialized program is produced. Chapter 7 has already addressed program generator generation (using program specialization to improve the speed of a specializer); we suggest some other techniques here.

Respecialization and Prespecialization

Current specializers, both online and offline, often perform the same reductions many times when they could perform them just once. That is, if we specialize a function f on an argument vector a, then on another argument vector $a' \sqsubset a$, we redo the entire symbolic evaluation of f on the new argument vector. However, we already know that all of the reductions made under a can also be made under a', so it would make more sense to start with the specialized function constructed on a and make additional reductions using that as a starting point. Such a *respecialization* mechanism might save a lot of effort.

Continuing in this vein, if we are going to specialize a function f on a number of different argument vectors a_i, it might make sense to first specialize f on some a' more general than all of the a_i, and use that as a starting point, even if the specialization on a' will not appear in the final specialized program. For example, pre-specializing f on $\top_{Val}$ eliminates all nonrecursive and statically bounded recursive procedural abstractions in the body of f by unfolding them. Since the cost of FUSE's recursion detection scheme is at least quadratic in the number of procedure calls performed by the specializer, reducing the number of procedure calls to be investigated by all of the specializations on the a_i reaps large benefits. Such strategies may also be useful in the offline world, for different reasons—the pre-specialization may duplicate code in the source program, partially mitigating the affects of a monovariant (or limited-polyvariant) binding time analysis.

Borrowing Offline Techniques

Online specialization does not preclude the use of static analysis techniques; the term "online" means that the specializer is *allowed* to use dynamic analyses, but not that it is *required* to do so in all cases. We see two ways in which offline techniques can be useful without any loss of accuracy.

First, we can use static analyses that correctly predict specialization-time information. For example, an online specializer could use an accurate binding time analysis (one that returns "static," "dynamic," or "unknown binding time" instead of lumping "unknown binding time" in with "dynamic") to avoid some amount of data structure traversal.[5] Similarly, if a static analysis can prove that certain parameters assume a finite number of values at specialization time, the specializer can optimize its termination mechanism; *e.g.,* not run induction analyses on those values, and compare these values first when testing for equality of argument vectors in the call stack. Other potentially useful analyses include liveness analyses (where we only use the fact that a value or binding is provably alive or provably dead) and control flow analyses (if we can prove that a particular function or closure can't invoke itself recursively, we can avoid having to track it in the termination mechanism).

Second, we can run the usual static analyses at specialization time (instead of prior to specialization) to avoid inaccuracies due to the monovariant nature of the analyses. For example, once the specializer has decided to unfold a particular procedure call, it might be worthwhile to run a binding time analysis on the body of the procedure, since we expect that all of the reduce/residualize decisions in the body will be made many times (once per iteration of the loop being unfolded) otherwise. At the very least, we will be able to determine that all operations involving a completely static induction variable will be reducible on all iterations, and that reduce/residualize decisions on circulating (nonchanging) parameters will the same on all iterations. Taken to the limit, this approach is basically specialization-time, dynamic "compilation" of the code of the specializer.

8.2.3 Automation

To obtain good results, most partial evaluators require some amount of help from the user, either in the form of explicit annotations or in the form of binding time transformations

[5] This isn't actually all that useful unless self-application or some other form of static precomputation based on binding times (such as the "action trees" of [25]) is used.

performed on the source program. FUSE has made some progress on this front; the termination mechanisms of [115] provide termination without being excessively conservative, and the mechanisms of Chapters 4 and 5 reduce the need for binding time transformations. However, there is much work still to be done.

The most important need for automation lies in the area of termination: while most end users may accept the need for explicit pragmas or directives to achieve optimal performance, they are unlikely to tolerate an optimizer that doesn't terminate under all circumstances. Most abstract interpreters and dataflow analyzers solve the termination problem either through limiting the value domain or by limiting polyvariance, often both. Given that a good partial evaluator must transcend both these restrictions to achieve good performance, termination becomes a very hard problem. Recent work on termination, in static and dynamic contexts, can be found in [54] and [70], respectively. Given the existence of the halting problem, users will always have to choose between guaranteed termination and optimal performance. We believe that a good system should provide acceptable residual programs under a termination guarantee, and allow the user to provide additional information in cases where the termination mechanism is found to be overly conservative.

8.2.4 Integration

Although partial evaluation-like optimizations, such as constant folding, copy propagation, and inlining, are often found in true "native-code" compilers, the vast majority of partial evaluation research has been performed under source-to-source frameworks. Although this is convenient for exploring the space of optimizations, it limits the degree of optimization in several ways.

First, some optimizations, although detectable by a partial evaluator operating on source code, cannot be expressed in the source language. For example, it might be easy for the specializer to determine that closures constructed from a particular `lambda` expression are only passed downward, never upward. This information would be useful to an underlying Scheme compiler, but there is no way to express this information in the Scheme language, and thus the underlying compiler must discover this information itself. In this case, it might well be able to, but in other more specialized cases, it might not.

Second, even though a particular transformation might be expressible in source code, the decision about whether to perform it may require additional knowledge about the underlying virtual machine. Consider *arity raising* [90], in which tuple-valued parameters are "spread"

into multiple distinct parameters. This optimization is probably beneficial only if there are sufficient free registers to hold the distinct parameters; otherwise, it's probably cheaper to leave the tuples in place on the stack or heap and just pass pointers. Unfortunately, a source-to-source partial evaluator knows nothing about the register set of the underling machine, or even the Scheme compiler's register allocation strategy. One might choose to rely on an underlying compiler that automatically performs "tupling" and "de-tupling" operations, but, in that case, there would be little point in having the arity raising optimization in the partial evaluator in the first place.

Finally, some optimizations may be neither detectable nor expressible in the source language, such as optimizing address calculations involving arrays or structures at partial evaluation time. Some of this can be solved by making such operations more explicit in the source language (*e.g.,* explicit environment conversion or cell conversion) but this is not always possible.

Thus, we believe that it would be worthwhile to investigate both (1) implementing a partial evaluator at some intermediate language level in a "real" compiler, and (2) implementing a partial evaluator that can cooperate with an "open" compiler using explicit pragmas or even a meta-level protocol [75, 89].

8.2.5 Applications

This work has concentrated largely on methods rather than on applications. We believe that work on applications is of great importance, for two reasons. First, if partial evaluation is to be popularized, it will be important to demonstrate its applicability to a wide range of problems, not just interpreters and their close relatives. Luckily, contemporary research is moving into newer domains, such as incremental computation [110] and operating systems [30]. We believe that meta-object protocols [72, 75] and reflective language implementations [36, 106] will also provide significant opportunities for partial evaluation.

Second, in some domains, partial evaluation can provide large performance benefits using very simple specialization strategies. For example, the work in [10] used a very simple partial evaluator that produced only straight-line code (*i.e.,* had no procedure specialization or termination mechanism, and could not produce residual loops), yet obtained quite large speedups—the insight was to specialize only the body of an inner loop in a numeric computation. Similar insights for other problem domains might yield good results.

Appendix A

Formalisms

In this Appendix, we more formally describe some of the concepts and mechanisms from Chapters 3, 4, 5, and 6. Our treatment is divided into six sections. We begin (Section A.1) by describing some basic properties of the specializer, and continue (Section A.2) with a discussion of the general framework used by our arguments. The remaining sections (Sections A.3 through A.6) describe properties of the base specializer, the return value fixpointing mechanism, the control-flow-based iterative specialization mechanism, and the re-use mechanism, respectively.

A note to the reader: We recommend that readers familiarize themselves with the contents of Chapter 3 before proceeding, as we make heavy use of terminology introduced in that chapter.

A.1 Basics

This section describes basic properties of the specializer's data structures and operations. We begin with the type system (Section A.1.1) and primitive operators (Section A.1.2). Because correctness arguments in later sections will require monotonicity properties of the specializer, Section A.1.3 defines partial orderings environments and closures. We conclude by describing some useful properties of the specializer's instantiation and termination mechanisms.

A.1.1 Type System

All program specializers manipulate specialization-time values which denote collections of runtime values (in the case of FUSE, Scheme values). As described in Section 3.1.2, FUSE's uses *value specifications* drawn from a type system for this purpose. Different versions of FUSE use different systems, and this doesn't affect any of the basic algorithms in the specializer. However, we do need to set some "ground rules" for choosing a type system.

First, we require that the elements of *Val* form a lattice, or CPO.[1] That is, we must be able to define a partial ordering relation $\sqsubseteq$ on pairs of elements of *Val* such that any two elements have a unique least upper bound. We do not require that *Val* be finite, as this would prohibit us from representing all Scheme values (such as integers or pairs, of which there are an infinite number). We also do not require that *Val* be of finite height, but merely that the lattice contain no infinite ascending chains of values, so that the usual iterative procedure for finding least fixed points will work.[2] If we choose the usual partial order on pairs ($\langle a, b\rangle \sqsubseteq \langle a', b'\rangle \Leftrightarrow a \sqsubseteq b \wedge a' \sqsubseteq b'$), then this can be easily accomplished by using *strict products* when constructing pairs and environments. That is, if structured types are not allowed to contain $\bot_{Val}$, then any non-bottom lattice point can lie at most a finite number of levels below $\top_{Val}$.

Second, we need to establish a relationship between value specifications and the set of runtime values they denote. As in most abstract interpretation systems, we do this with *abstraction* and *concretization* functions. The abstraction function A takes a set of Scheme values and returns a value specification denoting at least that many values, while the concretization function C takes a value specification and returns the set of Scheme values it denotes.

$$A : (\mathcal{PS}\ SchemeValue) \rightarrow Val$$
$$C : Val \rightarrow (\mathcal{PS}\ SchemeValue)$$

These mappings must preserve several properties. Mapping a set of concrete values to an abstract value and back again must preserve all of the concrete values in the original collection; it's permissible for the resultant collection to be larger, but it must contain all

[1] Not all versions of FUSE require that *Val* have a unique "bottom" element, but there's no harm in always requiring one.

[2] We can weaken this even further by allowing infinite ascending chains, but requiring that the specializer never explore them. We won't try this here.

of the original values. This makes sure we never "lose" any values when we approximate a set of values with a single value specification.

$$\forall x \in (\mathcal{PS}\ SchemeValue)\,[x \subseteq (C\,(A\,x))]$$

The partial order defining the type lattice must be a conservative abstraction of the subset operator on collections of concrete values; that is,

$$\forall x, y \in Val\,[x \sqsubseteq y \Rightarrow (C\,x) \subseteq (C\,y)]$$

This is important because it lets us approximate concrete subset relations in the abstract world. For example, when FUSE specializes a procedure on some argument v, it establishes (by construction) that the specialization can be safely applied to any values in $C(v)$. The relation above means that any abstract value $v' \sqsubseteq v$ will denote only values in $C(v)$, allowing FUSE to decide when a specialization is or is not applicable to a particular abstract value (without always having to build a new one).

The least upper bound operation on abstract values must be a conservative abstraction of the union operator on collections of concrete values:

$$\forall x, y \subseteq (\mathcal{PS}\ SchemeValue)\,[x \cup y \subseteq (C\,((A\,x) \sqcup (A\,y)))]$$

This allows the specializer to safely approximate unions of sets, which happens, for example, when an `if` statement is not reducible at specialization time, forcing the specializer to compute a single approximation which denotes all the values that could be returned by either arm. (Given the relationship between the inclusion operators and upper bounds, this is really saying the same thing as the previous relation.)

These properties are easy to achieve; one very simple system (used in some non-partially-static program specializers) abstracts all singleton concrete values to themselves, and all sets of concrete values to $\top_{Val}$, with the obvious concretization function. The type systems described in Chapter 2 aren't much more complicated; they just add structured abstract values with (possibly typed) "holes" denoting infinite collections of values (such as "all integers" or "all values").

A.1.2 Primitives

FUSE can be thought of as an abstract-interpretation-based type inference system, plus a code generator which makes use of the type information. Like other such systems, FUSE executes *abstract primitives* on *abstract values*, and expects that the results will conservatively approximate the results of running the corresponding *concrete primitives* on the corresponding *concrete values*. More formally, for any Scheme primitive p, FUSE implements an abstract primitive $\hat{p}$ such that[3]

$$\forall x \in Val \left[\{(p\, x') \mid x' \in (C\, x)\} \subseteq (C\, (\hat{p}\, x))\right] .$$

We also require that the specializer's abstract primitives be monotonic. That is, executing a primitive on more general inputs (points higher in the lattice) should yield an equivalent or more general result (points higher in the lattice), but never a more specific result (points lower in the lattice). This is just common sense; it should never be the case that knowing *less* about the inputs to a primitive should allow you to deduce *more* about the result. Formally, for all operators $\hat{p}$,

$$\forall x, x' \in Val \left[x \sqsubseteq x' \Rightarrow (\hat{p}\, x) \sqsubseteq (\hat{p}\, x')\right] .$$

A.1.3 Partial Orderings

FUSE has a replaceable type system module; with a little programming, the user can "plug in" any lattice *Val* whatsoever, provided that the appropriate partial ordering and primitive operations are defined (the system performs much of the effort of computing the induced partial order, projection functions, and primitive operations on *Val'*). However, when we attempt to exploit monotonicity properties of the specializer, we will make use of partial orders over the specializer's other data structures: environments and caches, which are not part of *Val*[4] and thus do not have a user-defined partial ordering (though, as we shall see, the partial ordering is strongly determined by the one chosen for *Val*). For now, we consider only partial orderings on environments, which are the same for all versions of the specializer.

[3] This formulation is for unary primitives but its extension to arbitrary arity should be clear.

[4] Environments may or may not be part of *Val*, depending on whether the particular version of FUSE supports higher-order constructs. Our arguments require that we define a partial order on environments even for the first-order-only case.

Partial orderings on caches will be introduced only when we need them, in Section A.4.

As we described before (*c.f.* Section 3.2.2), an environment can be viewed as a (partial) mapping from Scheme identifiers to symbolic values, *e.g.*[5]

$$Env = Var \rightarrow Sval$$

Since we will only ever be comparing environments binding exactly the same identifiers, it's easy to define a suitable partial order using the standard "lifting" of a partial order on values to a partial order on functions:

$$\forall r, r' \in Env\,[r \sqsubseteq r' \Leftrightarrow \forall v \in Dom\, r\,[(\textit{value-projection}\ (r\, v)) \sqsubseteq (\textit{value-projection}\ (r'\, v))]]\ .$$

Once we have an ordering on environments, the ordering on closures follows. Two closures are comparable only if they were constructed from the same `lambda` expression; if so, the ordering is induced by the ordering on their environments. That is

$$\forall c, c' \in Closure\,[c \sqsubseteq c' \Leftrightarrow (\textit{closure-expression}\, c) = (\textit{closure-expression}\, c) \wedge (\textit{closure-environment}\, c) \sqsubseteq (\textit{closure-environment}\, c)]\ .$$

A.1.4 Instantiation

Chapter 3 described the *instantiation* operation, which, given a value specification and a code expression, constructs a symbolic value; it gave an example, but no formal definition. In this section, we give a definition for a simple domain of value specifications, and two properties which must apply to *all* implementations of instantiation.

The instantiation procedure takes a value specification (element of *Val*) and a graphical code expression (element of *Code′*), and returns a symbolic value (element of *Sval*). For scalar value specifications, it simply constructs a new symbolic value on the value and the code (since scalars are the same in *Val* and *Val′*). For pair values, things are slightly more complicated: to obtain a specification in *Val′*, we must instantiate the specifications of the `car` and `cdr` on the original code, wrapped with appropriate accessors. Closure values must have their environments instantiated, where (as described in Section 3.2.2) this means instantiating each closed-over value on a unique symbol generated from the corresponding variable name.

[5] Correct programs will have no unbound variable references, so the partial nature of the mapping is not an issue.

```
instantiate : Val → Code′ → Sval
instantiate v c =
  cond (scalar? v) → ⟨ v , c ⟩,
       (pair? v) → s
          whererec s = ⟨ ⟨ s1 , s2 ⟩ , c ⟩
              where s1 = (instantiate v1 (PRIMCALL car s))
                    s2 = (instantiate v2 (PRIMCALL cdr s))
                 where ⟨ v1 , v2 ⟩ = v,
       (closure? v) → s
          where s = ⟨ ⟨x_i^n, b, env′⟩ , c ⟩
              where env′ = [(instantiate v_i (VAR (gensym var_i)))/var_i]^n
                  where ⟨x_i^n, b, [v_i/var_i]^n⟩ = v
```

Figure A.1: An instantiation procedure for scalars, pairs, and closures

---

A sample implementation of the instantiation procedure is shown in Figure A.1. In that code, and throughout the remainder of this Appendix, value specifications of closures are represented as triples of the form $\langle formals, body, environment \rangle$.

The partial evaluator relies on two properties of the instantiation procedure:

Property 1: *Instantiation preserves value specifications; when the value specification projection is applied to the new symbolic value, we always get the original value specification back:*

$$\forall v \in Val, c \in Code' \left[(value\text{-}projection\ (instantiate\ v\ c)) = v \right].$$

Because value specifications for structured types are instantiated by recursively applying the instantiation procedure to subcomponents of the value specification (and suitable code expressions), this property is also true for all subcomponents of the value specification.

Property 2: *The code expressions generated by the recursive invocations of the instantiation procedure are "correct" in the following sense: if we instantiate $v \in Val$ on $c \in Code'$, yielding a symbolic value $\langle v' , c \rangle$ where $v' \in Val$, then for any accessor operation a (*e.g., `car`*, or* `caddr`*), the code expression obtained by applying the accessor to the value specification attribute v' will be equivalent (*i.e., *return the same values at runtime) as the expression obtained by composing the accessor with the original code c. Loosely speaking, $(a\ v') \approx (a \circ c)$.*

This should be obvious for the outermost symbolic value (since the empty sequence of accessors returns the original code expression), and for pairs (because the code for each subcomponent is simply the code for the enclosing pair, "wrapped" with the appropriate accessor). Because applying a closure and accessing a variable within its environment returns only a variable name ("forgetting" the accessor sequence used to reach the closure), our correctness property applies only if the specializer guarantees that the appropriate bindings will be performed. Because our method for doing this (arity raising) works only for closures that are passed downward in the residual program, we forbid "upward" generalization of closures out of residual `if` expressions, since the generalized value could not be correctly instantiated (*c.f.* Page 64).

A.1.5 Termination

Perhaps our biggest assumption regards termination. None of our arguments will prove that the specializer terminates; indeed, they will be correct only when it does. As we have done throughout this dissertation, we assume an appropriate termination mechanism, and rely on its properties (the properties described below are true of the termination mechanisms described in Section 3.3.4, so they are indeed implementable).

The main property of a termination mechanism is that it causes the specializer to terminate. That is, it allows the specializer to construct only a finite number of specializations, each of finite size. It accomplishes the latter by limiting unfolding (forcing specialization), and the former by limiting polyvariance (abstracting argument specifications). We take this as a given; our interest here is limited to some properties of this decision-making process.

Our idealized specializer interacts with its termination mechanism through two functions. The first, *unfold?* takes a function name and an argument specification[6] and returns

[6]The function name and argument specification are not, in general, sufficient information to allow this decision to be made; *unfold?* relies on annotations and/or internal state which we will not make explicit here. See Section 3.3.4 or [115] for more details.

a boolean value which is true if the function call should be unfolded, and false if it should be residualized. Whenever *unfold?* returns false, the specializer calls *get-general-args* on the function name and argument specification, and receives a new argument specification to use in deciding which specialized version of the function should be invoked by the residual call (*i.e.,* the new argument specification is used for searching the cache and/or constructing the body of the specialized procedure).

Property 3: *The unfold/specialize decision is monotonic with respect to argument specifications. That is, if* (*unfold?* f a) *returns* false *(indicating that the call should be residualized) for some function name* f *and argument specification* a*, then all calls* (*unfold?* f a') *where* $a \sqsubseteq a'$ *must also return* false.

An intuitive justification for this restriction is the following. When we know the value of a loop's induction variable(s), we unfold it; when we don't, we construct a residual loop. It should never be the case that knowing *less* about the values of the loop's induction variable(s) should allow us to unfold the loop (*i.e.,* deduce the number of iterations at specialization time) when *more* information would not; this wouldn't make any sense.

Property 4: *When, at any particular call site, the termination mechanism chooses to residualize, the specialization is constructed on an argument specification at least as general as that on which unfolding would have been be performed. That is, if* (*get-general-args* f a) *returns an argument specification* a' *for some function name* f *and argument specification* a*, then it must be the case that* $a \sqsubseteq a'$*. Furthermore, get-general-args is monotonic with respect to its argument specification;* i.e., $a \sqsubseteq a' \Rightarrow$ (*get-general-args* f a) $\sqsubseteq$ (*get-general-args* f a').

Intuitively, the specialization must be sufficiently general to be applicable at all call sites, so it can't be constructed on an argument specification more specific than at any particular call site (such as the one passing values a). This part of the property is true of all specializers. Furthermore, if, given an argument specification, we must discard some amount of information to assure termination (*i.e.,* by discarding statically known values that vary from iteration to iteration, but are not themselves induction variables[7]), it should not be the case that, given less information, we should have to discard more.

[7] These are often called "counters" in the literature [70].

A.2 Methods of Argument

Before going on to describe the implementation of the specializer and its properties, we take a moment to describe the basic structure of the arguments we will be making, and the assumptions underlying them.

A.2.1 Evaluators

First, some of our arguments will refer to the semantics of the source (kernel Scheme) and specialized (graphical) programs. That is, we will want to show that, for particular input values, executing specialized programs gives the same result as executing source programs. Rigorous proofs would require that we define the semantics of the input and output languages, and that we somehow use these to relate the meanings of the input and output programs. We will not do this; instead, we will show that, locally, at each reduction step, the specializer preserves the meaning of the input program, and that the composition of these steps also preserves meaning.

Still, our statements of various properties will refer to evaluators for source and graph programs, so we devote a few words to describing them. The evaluator for source programs, $\mathcal{E}$, takes a source expression, a Scheme environment mapping the expression's free variables to Scheme values, and the entire program text,[8] and returns a Scheme value. We will not describe this evaluator formally; any interpreter obeying the commonly-understood semantics of functional Scheme (*i.e.,* a subset of the language described in [88]) will do.

Evaluation of graphical programs is less obvious. We want to evaluate a graph node g in an environment mapping all of the identifiers appearing in `VAR` nodes free in g (*i.e.,* `VAR` nodes reachable from g but not enclosed in a `LAMBDA` node binding that same `VAR` node) to Scheme values, and get back a Scheme value as the result.

For first-order programs, the primary difference between source and graph programs (elements of the domain *Code* of Figure 3.4) is that graph programs are not trees; a single graphical expression may be referred to by many other nodes, including itself; thus `let` and `letrec` forms are are not needed, nor is the program text required (since graph nodes refer directly to the text of any functions they may call). All other forms (constants, variables, conditionals, applications, and `lambda` expressions) retain their usual meanings. As far as evaluation is concerned, these changes make no difference—if we simply run the source

[8] Which is needed for looking up top-level function definitions.

program evaluator $\mathcal{E}$ on a node, an environment mapping identifiers to Scheme values, and a dummy program, we will get the correct Scheme value as the result. Of course, such a naive evaluator will execute multiply-referenced nodes multiple times, but in a functional language this makes no difference to the result.[9] Thus, we can think of the input and output languages as having the same semantics, and concern ourselves with the legality of local reductions made by the partial evaluator.

However, as we described in Chapter 3, residual programs aren't really graphs of expressions (elements of *Code*), but are embedded in symbolic values using the domain *Code'* of Figure 3.5. The difference is that a symbolic value has a value specification as well as a code expression. Given a symbolic value with a ground (*i.e.,* denotes a single Scheme value) value specification, the graph evaluator may return that ground value instead of evaluating the code expression; symbolic values with expression `no-code` are required to have ground value specifications.

Higher-order programs add a complication, namely the arity raising of procedures with respect to closure-valued arguments. Any closure which is both created and applied within a single specialization poses no problems, since all of the closure's bound variables will evaluate to symbolic values whose code expressions will reference only formal parameters of the specialization. However, if a procedure is specialized on a closure-valued argument, we *instantiate* all of the closed-over values in the closure's environment on new names, and consider those names part of the specialization's formal parameters.[10] When we unfold the instantiated closure, any references to closed-over variables in the closure's body will return symbolic values whose code expressions reference the new formal parameter names. Thus, specializing an expression in an environment may yield a residual expression (graph node) that references variables that aren't bound by the specialization-time environment, but that will be bound at runtime.

[9] FUSE's code generator, which converts graph programs back to the source language, performs hoisting to avoid such inefficiencies. Reasoning directly about graph programs saves us from having to define and prove properties of the code generation process. It is sufficient to say that applying the source program evaluator to the output of the code generator always yields the same result as applying the graph evaluator to the input to the code generator.

[10] Because a procedure might be specialized with respect to multiple closures which close over the same variable names, we must instantiate the closed-over values on new, unique names, effectively alpha-converting the specialized program as it is constructed. Another way to accomplish this without the need to construct new names is to use the identity of `VAR` nodes in the graph for environment lookup, rather than the identity of the identifier names contained in the `VAR` nodes; this is what the actual implementation of FUSE does. Also, there is no need to create formal parameters for ground-valued closed-over variables.

If e is the specialization-time environment

$$\begin{array}{l}\{\ \texttt{a} \mapsto \langle\, \top_{Number}\ ,\ \texttt{(VAR a)}\,\rangle \\ \quad \texttt{b} \mapsto \langle\, \langle\ \texttt{(c)},\ \texttt{(+ c d)},\ \{\texttt{d} \mapsto \langle\, \top_{Integer}\ ,\ \texttt{(VAR new-d)}\,\rangle\}\rangle, \\ \qquad\quad \texttt{(VAR b)}\rangle \\ \}\end{array}$$

then specializing the application `(b a)` in e yields the specialized program

```
(PRIMOP + (VAR a) (VAR new-d))
```

If we take some runtime instance e' of e, such as

$$\begin{array}{l}\{\ \texttt{a} \mapsto \textit{3.14159} \\ \quad \texttt{b} \mapsto \langle \texttt{(c)},\ \texttt{(+ c d)},\ \{\texttt{d} \mapsto \textit{42}\}\rangle \\ \}\end{array}$$

we can't run our specialized program in this environment (or any other instance of e, because the variable `new-d` is not bound. If we compute $e'' = (\textit{spread-env}\ e\ e')$, yielding

$$\begin{array}{l}\{\ \texttt{a} \mapsto \textit{3.14159} \\ \quad \texttt{b} \mapsto \langle \texttt{(c)},\ \texttt{(+ c d)},\ \{\texttt{d} \mapsto \textit{42}\}\rangle \\ \quad \texttt{new-d} \mapsto \textit{42} \\ \}\end{array}$$

running the specialized program in e'' will return the desired value.

Figure A.2: Example of *spread-env*

$SchemeValue$			Scheme values
$SchemeEnv$	$=$	$Var \rightarrow SchemeValue$	Scheme environments
$\mathcal{E}$	$=$	$Exp \rightarrow SchemeEnv \rightarrow Pgm \rightarrow SchemeValue$	Evaluator for source programs
$\mathcal{E}'$	$=$	$Sval \rightarrow SchemeEnv \rightarrow SchemeValue$	Evaluator for graph programs

Figure A.3: Evaluator functions for source and graph programs

If we wish to equate source and specialized programs in such cases, we need a way to map these new variable references to the corresponding values. We do this by defining a new operator *spread-env* that takes a specialization-time environment e and a Scheme environment e' where $e' \in (C\,(\textit{value-projection}\,e))$, and returns an extended version of e'. It finds all closures c contained in symbolic values (including closures in pairs and in the environments of closures) bound to variables in $v\ e$. For each symbolic value s in c's environment with code expression `(VAR` i`)` where identifier i is not bound in e, *spread-env* dereferences v in the corresponding closure environment in e', returning some value x, then adds a binding to e' mapping i to x.[11] Figure A.2 gives a small example.

Thus, when we invoke the graph evaluator $\mathcal{E}'$ on a program, and a Scheme environment, the Scheme environment must always be obtained by applying *spread-env* to a specialization-time environment and one of the Scheme environments it denotes.

Another complication with higher-order programs involves comparing source and specialized programs where the return values might be closures; this would require that we define an operational equivalence relation on closures. Although FUSE does allow programs to return closures, we will avoid the additional mechanism by simply prohibiting programs from returning closures; thus, we need only compare atomic and pair values, for which the notion of equality is unambiguous.

Thus, we will define the graph evaluator $\mathcal{E}'$ as a function which takes a symbolic value and an environment mapping variable names to Scheme values, and returns a Scheme value. The return value is the same as that returned by the source program evaluator $\mathcal{E}$ applied to the code projection of the symbolic value, modulo the substitutions for ground values described above. Figure A.3 summarizes the signatures of the evaluators.

[11] We are assuming there are no name clashes between the names i and any other name bound in e or e'; this is a valid assumption because the specializer guarantees this property when performing the instantiation operations creating the closures c (*c.f.* previous footnote).

A.2.2 More on Equality

We have already seen one technical issue related to the difference between the runtime environments of the source and specialized programs; namely, that arity raising may introduce new names that are not bound in either the source runtime or specialization-time environments, but that will be bound in the runtime environment of the specialized program. We used the function *spread-env* that used the specialization-time environment to provide a relation between the runtime environments of the source and specialized programs in the presence of arity raising.

There is another issue in relating runtime environments: namely, that it is sufficient for the runtime environment of the specialized program to model the runtime environment of the source program at all specialization points (*i.e.,* the actual parameters passed to source procedures and specialized procedures are the same modulo the arity raising described above). Indeed, in a graphical program, the *only* variable names (`VAR` nodes) are those referring to formal parameters of specializations; all variable references due to `let`-binding and unfolded procedure applications are gone. Thus, it only makes sense to compare source and residual expressions by evaluating them in "equivalent" environments if the residual expression is the body of a specialized procedure.

However, for the purposes of proving correctness via induction, we will have to show the correctness of arbitrary residual expressions. We can't use (concretizations of) the specialization-time environment for this purpose, since the specialization-time environment in which some expression is specialized will not, in general, contain bindings for all of the formal parameters of the enclosing specialization.[12] Thus, when considering the specialization of a particular source expression, we must consider not only the specialization-time environment in which the expression is specialized, but also the specialization-time environment defining any identifiers appearing free in any code expressions—namely, the specialization-time environment in force when we began specializing the body of the enclosing specialization. To make proofs possible, we will "instrument" our idealized implementation of the partial evaluator to track this environment at all times; such tracking is of course

[12] Consider specializing `f=(lambda (a b) ...)` on some values, which binds `a` and `b` to symbolic values with code fields `(VAR a)` and `(VAR b)`, respectively. If, inside the body of `f`, we unfold `g=(lambda (c) ...)` on `(+ a b)`, then `g`'s body will be evaluated in an environment where `c` is bound to a symbolic value with code `(PRIMOP + (VAR a) (VAR b))`, but in which `a` and `b` are unbound. The correctness of the code bound to `c` can be established only in the context of the environment binding `a` and `b`; the binding for `c` is largely irrelevant because it does not appear in the residual program.

unnecessary in the actual implementation (*i.e.,* the necessary invariants are preserved by construction; we need the explicitness only to show that this is the case). This is the purpose of the formal parameter *senv* in the implementation of Section A.3.1.

A.2.3 Induction Technique

Users typically view the program specializer through its top-level interface, *PE-program*, which takes, among other things, a *program*, and returns a new *program*. Because this entry point is invoked only once, it's not very useful for constructing induction hypotheses. Instead, we will consider properties of *PE*, which specializes an *expression* in the context of an environment, cache, etc., and returns a *symbolic value*. As we shall see in Section A.3.1, both the specializer for programs, *PE-program*, and the specializer for functions *PE-spec*, are implemented in terms of *PE*, so the desired properties will hold for them as well.

The method we will use for most of the arguments in this Appendix is induction over the length of the specialization process. At first, this may seem counter-intuitive—one might expect to perform structural induction over the source program or over the residual program. The first approach doesn't work because of polyvariance; that is, the specializer may examine a particular source expression many times under different contexts, which means we can't directly relate specialization-time reductions to their positions in the source code.[13] The second approach doesn't work because walking a residual code graph doesn't tell you anything about the source program from which it was generated; because of the reduction of `let` expressions, unfolding of procedure calls, and copy propagation through data structures, a single residual code expression can be the specialization of *many* source expressions. When we add in dead code optimizations, we find that some specializations (*i.e.,* an addition that reduces to `4`, or even one that only reduces to $\top_{Number}$, but is only ever passed to `number?`) are completely consumed at specialization time—we need to reason about the correctness of those reductions too, but all trace of them is gone by the time we have a complete residual program.

[13] Actually, we could largely solve this problem by postulating an "oracle" which tells us how many times each source function will be unfolded, and which call sites will construct/re-use which specializations. Then, if we think of the specializer as just performing beta-substitution and delta-reductions on this "pre-unfolded" source program, we could make arguments via structural induction on the source. However, this relies just as much on the termination mechanism as does our induction method, and introduces extra conceptual overhead, so we won't use it.

Instead, we will perform induction over the number of steps remaining in the specialization of a particular expression under the operational specification of our specializer given in Section A.3. Thus, our arguments will be valid only for cases in which the specializer terminates, but that's fine since (1) we explicitly assume that the termination heuristics work, and (2) non-terminating specialization processes don't produce results to reason about anyway.

A.3 Basic PE

In this section, we present a simplified implementation of base FUSE, and argue that it produces correct code.

A.3.1 Description of the Specializer

In this section, we present a simplified implementation of the specializer which we will refer to when describing various properties in later sections. Our description is operational in nature, and can be thought of as a non-standard interpreter for Scheme programs. In the interest of brevity, we will use a denotational-semantics-like programming language[14] with pattern matching (indicated by double brackets [[]]) and destructuring binding constructs.

Most of the specializer's work is accomplished by the function *PE*, which specializes an expression given:

- an environment mapping its free variables to symbolic values,
- a cache containing (possibly incomplete) specializations,
- the text of the entire program (so that functions can be looked up), and
- the environment in effect when we began constructing the current specialized function (never referenced by the implementation, present only for proofs).

and returns

- a symbolic value (denoting the specialization of the expression), and
- a (possibly updated) cache, reflecting any specialized functions constructed during the specialization of the expression.

[14] This doesn't mean our description is denotational; it isn't. We're just using the same syntax because it's a familiar one.

Source Code (*c.f.* Figure 3.2)

Exp			source code expressions
Fcn			user function names
Prim			primitive names

Graph Code (*c.f.* Figure 3.5)

Code			graphical code expressions
Code'			*Code* with embedded *Sval*s
Code-fcn	=	$Sval^* \times Sval$	specialized functions

Specializer (*c.f.* Section 3.2)

Val			value specifications
Val'			*Val* with embedded *Sval*s
Sval	=	$Val' \times Code'$	symbolic values
Env	=	$Var \rightarrow Sval$	specialization-time environments
Cache	=	$Fcn \rightarrow Val^* \rightarrow Code\text{-}fcn$	specialization caches

Figure A.4: Domains used by the specializer

PE-program	:	$Pgm \rightarrow Fcn \rightarrow Val^* \rightarrow Code\text{-}fcn$
PE	:	$Exp \rightarrow Env \rightarrow Cache \rightarrow Pgm \rightarrow Env \rightarrow Sval \times Cache$
PE-seq	:	$Exp^* \rightarrow Env \rightarrow Cache \rightarrow Pgm \rightarrow Env \rightarrow Sval \times Cache$
PE-prim	:	$Prim \rightarrow Sval^* \rightarrow Sval$
PE-spec	:	$Fcn \rightarrow Val^* \rightarrow Cache \rightarrow Pgm \rightarrow Code\text{-}fcn \times Cache$
value-projection	:	$Sval \rightarrow Val$
instantiate	:	$Val \rightarrow Code' \rightarrow Sval$

Figure A.5: Signatures of major functions in the specializer

PE makes use of several helper functions, including *PE-seq*, which evaluates several expressions in sequence (single-threading the cache), and returns the resultant symbolic values and final cache, and *PE-prim*, which applies primitive operators to symbolic values, returning symbolic values. The function *PE-spec* constructs specialized versions of functions; given a function name and a sequence of value specifications, it returns a residual function object (a *Code-fcn*) and a cache.

In the remainder of this section, we examine the implementation of these functions in greater detail.

Special Forms

We begin with the specialization of constants, which is quite simple: we merely execute the abstraction function to map the source text of the constant to a value specification, then instantiate that value specification on `no-code`. The cache is not altered.

$$PE\,[\![\texttt{(CONST}\ k\texttt{)}]\!]\ env\ cache\ pgm\ senv$$
$$= \langle (instantiate\ (constant\text{-}to\text{-}Val\ k)\ \texttt{no-code}),\ Cache\rangle$$

Variables are even simpler; we just look them up in the environment.

$$PE\,[\![\texttt{(VAR}\ v\texttt{)}]\!]\ env\ cache\ pgm\ senv$$
$$= \langle (env\ v),\ cache\rangle$$

The specializer handles `let`-bindings by evaluating each of the initializing expressions in the current environment, extending the environment with the results, then evaluating the body in the new environment.

$$PE\,[\![\texttt{(LET}\ (x_i\ e_i)^n\ e\texttt{)}]\!]\ env\ cache\ pgm\ senv$$
$$= PE\ e\ env'\ cache'\ pgm$$
$$where\ env' = env\,[s_i/x_i]^n$$
$$where\ \langle s_i^n, cache'\rangle = PE\text{-}seq\ e_i^n\ env\ cache\ pgm\ senv$$

Conditionals are evaluated as follows. First, the predicate expression is evaluated; if it is known to be true or false, the appropriate arm is evaluated, and the corresponding symbolic value is returned. Otherwise, both arms are evaluated and the returned symbolic value is

constructed by instantiating the least upper bound of the arms' value specifications[15] on a residual `if` expression whose subforms are the symbolic values for the predicate, consequent, and alternative forms.

$$
\begin{aligned}
&PE\,[\![\texttt{(IF}\ e_1\ e_2\ e_3\texttt{)}]\!]\ env\ cache\ pgm\ senv\\
&\quad = cond\ (istrue?\ v_1) \rightarrow PE\ e_2\ env\ cache'\ pgm\ senv,\\
&\qquad\quad (isfalse?\ v_1) \rightarrow PE\ e_3\ env\ cache'\ pgm\ senv,\\
&\qquad\quad else \rightarrow \langle (instantiate\ v\ \texttt{(IF}\ s_1\ s_2\ s_3\texttt{)}),\ cache''\rangle\\
&\qquad\qquad\qquad where\ v = (value\text{-}projection\ s_2) \sqcup (value\text{-}projection\ s_3)\\
&\qquad\qquad\qquad\quad where\ \langle\langle s_2, s_3\rangle, cache''\rangle = PE\text{-}seq\ \langle e_2, e_3\rangle\ env\ cache'\ pgm\ senv\\
&\qquad\quad where\ v_1 = (value\text{-}projection\ s_1)\\
&\qquad\qquad where\ \langle s_1, cache'\rangle = PE\ e_1\ env\ cache\ pgm\ senv
\end{aligned}
$$

Calls to primitives are specialized by evaluating the arguments, then handing them off the the helper function *PE-prim*.

$$
\begin{aligned}
&PE\,[\![\texttt{(PRIMCALL}\ p\ e_i^n\texttt{)}]\!]\ env\ cache\ pgm\ senv\\
&\quad = \langle (PE\text{-}prim\ p\ s_i^n),\ cache'\rangle\\
&\qquad where\ \langle s_i^n, cache'\rangle = PE\text{-}seq\ e_i^n\ env\ cache\ pgm
\end{aligned}
$$

When faced with a call to a top-level function, the specializer first evaluates the arguments, then decides whether to unfold or to specialize (by calling the oracle *unfold?*). If the decision is to unfold, the body of the function is evaluated in an environment constructed by binding the function's formal parameters to the symbolic values obtained by evaluating the arguments. If the decision is to specialize, then the termination mechanism provides (via the function *get-general-args*) a new, possibly more general, vector of argument specifications, which are passed off to *PE-spec*, which constructs and memoizes (in the cache) the specialization. The final symbolic value consists of the value specification $\top_{Val}$ and a residual call to the specialization returned by *PE-spec*.

[15] At the risk of throwing away useful information, we could just instantiate the value specification $\top_{Val}$ instead of computing the least upper bound of the two arms' value specifications. This is done in some non-fixpointing versions of FUSE (because without good return value approximations, knowing the results of `if` expressions in loops isn't very useful anyway) and in the CFA version of FUSE (because it accepts only CPS programs, where `if` expressions are used only for control and not for their return values).

$$
\begin{aligned}
&PE\,[\![\texttt{(USERCALL}\ f\ e_i^n\texttt{)}]\!]\ env\ cache\ pgm\ senv\\
&\quad = cond\ (unfold?\ f\ v_i^n) \rightarrow (PE\ b\ [s_i/x_i]\ cache'\ pgm\ senv)\\
&\qquad\qquad\qquad\qquad\qquad\qquad where\ x_i^n = (formals\ f\ pgm),\ b = (body\ f\ pgm),\\
&\qquad\quad else \rightarrow \langle\langle \top_{Val},\ \texttt{(USERCALL}\ f'\ s_i^n\texttt{)}\rangle,\ cache''\rangle\\
&\qquad\qquad\qquad where\ \langle f', cache''\rangle = PE\text{-}spec\ f\ w_i^n\ cache'\ pgm\\
&\qquad\qquad\qquad\quad where\ w_i^n = (get\text{-}general\text{-}args\ f\ v_i^n)\\
&\qquad\quad where\ v_i^n = (value\text{-}projection\ s_i)^n\\
&\qquad\qquad where\ \langle s_i^n, cache'\rangle = PE\text{-}seq\ e_i^n\ env\ cache\ pgm\ senv
\end{aligned}
$$

The evaluation of `LAMBDA` forms is conceptually very simple: we just build a closure object containing the source code of the form and the current environment, so that the closure can later be unfolded or specialized. Problems arise when we consider specializing closures: some form of control flow analysis is needed to determine which closures indeed reach residual call sites and thus require specialization (as we discussed in Section 3.3.5, it's simply not practical to build specializations of all closures). In this formalization, we assume an "oracle" that tells us, at the time we execute a `LAMBDA` expression, whether the resulting closure will ever reach a residual call site, and construct a specialization (on $\top_{Val}$) if the oracle returns "yes." This gives the same effect as the real implementation, which waits until it sees a closure in a residual context before specializing it, but requires less (side-effect-intensive) machinery. Because we construct only one specialization per closure, there is no need for caching.

$$
\begin{aligned}
&PE\,[\![\texttt{(LAMBDA}\ x_i^n\ e\texttt{)}]\!]\ env\ cache\ pgm\ senv\\
&\quad = \langle(instantiate\ \langle x_i^n, e, env\rangle,\ spec\text{-}code),\ cache''\rangle\\
&\qquad where\ \langle spec\text{-}code,\ cache''\rangle =\\
&\qquad\quad cond\ \text{“}closure\ will\ not\ appear\ in\ residual\ context\text{”} \rightarrow \langle\texttt{no-code}, cache\rangle,\\
&\qquad\qquad \text{“}closure\ will\ appear\ in\ residual\ context\text{”} \rightarrow \langle\texttt{(LAMBDA}\ s_i^n\ b\texttt{)},\ cache'\rangle\\
&\qquad\qquad\quad where\ \langle b, cache'\rangle = (PE\ e\ env\ [s_i/x_i]^n\ cache\ pgm\ senv)\\
&\qquad\qquad\qquad where\ s_i^n = (instantiate\ \top_{Val}\ \texttt{(VAR}\ x_i\texttt{)})^n
\end{aligned}
$$

`ABSCALL` forms are similar to `USERCALL` forms, except that the head, as well as the arguments, must be evaluated. If the value specification of the head is a closure, then that closure is unfolded on the arguments (by evaluating its body in an appropriate extension of its environment); otherwise, the call is left residual.

$$
\begin{array}{l}
PE\ [\![\texttt{(ABSCALL}\ e_0\ \ e_i^n\texttt{)}]\!]\ env\ cache\ pgm\ senv \\
\quad = cond\ (isclosure?\ v_0) \rightarrow (PE\ b\ env''\ cache'\ pgm\ env'') \\
\qquad\qquad\qquad\qquad\qquad\qquad where\ env'' = env'[s_i/x_i]^n \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad where\ \langle x_i^n, b, env'\rangle = v_0, \\
\qquad\qquad else \rightarrow \langle (instantiate\ \top_{Val}\ \texttt{(ABSCALL}\ s_0\ \ s_i^n\texttt{)}),\ cache'\rangle \\
\qquad where\ \langle v_0, c_0\rangle = s_0 \\
\qquad\quad where\ \langle\langle s_0, s_1, \ldots, s_n\rangle, cache'\rangle = (PE\text{-}seq\ \langle e_0, e_1, \ldots, e_n\rangle\ env\ cache\ pgm\ senv)
\end{array}
$$

Sequences

The helper function *PE-seq* evaluates a sequence of expressions in the same environment, single-threading the cache through the evaluations, and returns the symbolic value produced by each expression, as well as the final cache.

$$
\begin{array}{l}
PE\text{-}seq\ [\![e_i^n]\!]\ env\ cache\ pgm\ senv \\
\qquad = \langle\langle s_1, s_2, \ldots, s_n\rangle, cache_n\rangle \\
\qquad\quad where\ \langle s_1, cache_1\rangle = (PE\ e_1\ env\ cache\ pgm\ senv) \\
\qquad\qquad\quad \langle s_2, cache_2\rangle = (PE\ e_2\ env\ cache_1\ pgm\ senv) \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\quad \langle s_n, cache_n\rangle = (PE\ e_2\ env\ cache_{n-1}\ pgm\ senv)
\end{array}
$$

Primitives

Applications of primitives are evaluated by the function *PE-prim*, which takes a primitive name, a tuple of symbolic values (the arguments), and produces a single symbolic value (the result). In keeping with our treatment of Section 3.3.2, we give sample implementations for `+`, `cons`, and `car`.

$$
\begin{array}{l}
PE\text{-}prim\ [\![\texttt{+}]\!]\ \langle s_1, s_2\rangle \\
\qquad = cond\ (number?\ v_1) \wedge (number?\ v_2) \rightarrow \langle (+\ v_1\ v_2),\ \texttt{no-code}\rangle, \\
\qquad\qquad else \rightarrow \langle \top_{Number},\ \texttt{(PRIMCALL + } s_1\ \ s_2\texttt{)}\rangle \\
\qquad\quad where\ \langle v_1, c_1\rangle = s_1,\ \langle v_2, c_2\rangle = s_2
\end{array}
$$

$$
\begin{array}{l}
PE\text{-}prim\ [\![\texttt{cons}]\!]\ \langle s_1, s_2\rangle \\
\qquad = \langle (cons\ s_1\ s_2),\ \texttt{(PRIMCALL cons } s_1\ \ s_2\texttt{)}\rangle
\end{array}
$$

$$
\begin{array}{l}
\textit{PE-prim}\ [\![\texttt{car}]\!]\ \langle s_1 \rangle \\
\qquad = \textit{cond}\ (\textit{pair?}\ v_1) \rightarrow (\textit{car}\ v_1), \\
\qquad\qquad \textit{else} \rightarrow \langle \top_{Val}, \texttt{(PRIMCALL car}\ s_1\texttt{)} \rangle \\
\qquad\quad \textit{where}\ \langle v_1, c_1 \rangle = s_1
\end{array}
$$

Specializations

The helper function *PE-spec* constructs, caches, and re-uses specializations. When invoked on a function name, an argument specification (a tuple of value specifications), a cache, and a program, it returns a residual function definition which is applicable to the provided argument specification, and a (potentially updated) cache. It operates by first checking the cache to see if an appropriate specialization already exists; if so, it returns it. Otherwise, it constructs a new specialization by looking up the function in the program and unfolding its body on an argument vector obtained by instantiating the argument specification on the function's formal parameter list. Because the specialization may be recursive, it is necessary to extend the cache with a *code-fcn* object for the new specialization *before* performing the unfolding, then update the dummy body of the *code-fcn* after the unfolding is complete. No code anywhere in the specializer ever looks inside a code expression, only inside value specifications, so the temporary presence of a dummy body in the *code-fcn* will pose no problems.

$$
\begin{array}{l}
\textit{PE-spec}\ f\ v_i^n\ \ \textit{cache}\ \ \textit{pgm} \\
\qquad = \textit{cond}\ (\textit{cache-result} = \bot_{Cache}) \rightarrow \\
\qquad\qquad \textit{begin} \\
\qquad\qquad\quad (\textit{set-code-fcn-body!}\ \textit{code}\ b) \\
\qquad\qquad\quad \langle \textit{code},\ \textit{cache}'' \rangle \\
\qquad\qquad\ \ \textit{where}\ \langle b, \textit{cache}'' \rangle = (\textit{PE}\ e\ \textit{env}\ \textit{cache}'\ \textit{pgm}\ \textit{env}) \\
\qquad\qquad\qquad \textit{where}\ \textit{env} = [s_i/x_i]^n \\
\qquad\qquad\qquad\quad \textit{where}\ \textit{cache}' = \textit{cache}\,[\langle f, v_i^n \rangle \mapsto \textit{code}] \\
\qquad\qquad\qquad\qquad \textit{where}\ \textit{code} = \texttt{(LAMBDA}\ s_i^n\ \textit{dummy}\texttt{)} \\
\qquad\qquad\qquad\qquad\quad \textit{where}\ s_i^n = (\textit{instantiate}\ v_i\ \texttt{(VAR}\ x_i\texttt{)})^n \\
\qquad\qquad\qquad\qquad\qquad \textit{where}\ x_i^n = (\textit{formals}\ f\ \textit{pgm}),\ e = (\textit{body}\ f\ \textit{pgm}), \\
\qquad\qquad \textit{else} \rightarrow \langle \textit{cache-result},\ \textit{cache} \rangle \\
\qquad\quad \textit{where}\ \textit{cache-result} = (\textit{cache}\ f\ v_i^n)
\end{array}
$$

Top Level

The top-level interface of the specializer takes a program, a goal function name, and an argument specification, and returns a residual function definition. This is easily accomplished by invoking *PE-spec*, as in

$$\begin{aligned} &\textit{PE-program pgm goal } v_i^n \\ &\qquad = f \\ &\qquad\quad \textit{where } \langle f, \textit{cache}' \rangle = (\textit{PE-spec goal } v_i^n \ [] \ \textit{pgm}) \end{aligned}$$

Unlike most other specializers, which must return a cache at top level so that code for specialized procedures defined in the cache can be emitted, FUSE does not return a cache. This is because the graph-structured residual code for the specialization of the goal function contains pointers to the residual code for all functions it (transitively) invokes; the code generator simply follows these links to determine which specialized procedures to emit.

A.3.2 Correctness

Our correctness argument will proceed in two steps. First, we will show that, at each step during the specialization process, the specializer computes correct approximations to the sets of values that could be returned by source code expressions executed under the given approximations to the values of their free variables. Then, we will show that, given these approximations, the specializer only performs reductions that would be made under all possible runtime executions (as constrained by the value approximations), and constructs residual versions of all non-reducible expressions, thus producing a correct residual program. The first subsection addresses the correctness property for type inference, while the next subsection describes the correctness property for residual code.

Correctness of Type Inference

What we want to show is that, at each step in the specialization process, the value specification in the returned symbolic value correctly approximates the set of values that could possibly be returned at runtime by the source code expression being specialized, provided that it is run in an environment compatible with the specialization-time environment with respect to which the specialization is being computed.

Property 5: *For any program $p \in Pgm$ defining all function symbols in expression $e \in Exp$, specialization-time environment $r \in Env$ defined on all free variables in e, cache $c \in Cache$, and "enclosing specialization environment" $h \in Env$, the value specification (value-projection s) of the symbolic value s returned by specializing e on r, c, p, and h denotes all values which could be returned by executing e under any Scheme environment denoted by r;* i.e., *for all Scheme environments $r' \in (C\,(\textit{value-projection}\,r))$*

$$(\mathcal{E}\,e\,r'\,p) \in (C\,(\textit{value-projection}\,s))$$
$$\textit{where } \langle s, c' \rangle = (PE\,e\,r\,c\,p\,h).$$

Proof: By induction on the number of calls to PE during the specialization of e.

Hypothesis: Property 5 is true for e', r', c', p, and h' where executing $(PE\ e'\ r'\ c'\ p\ h')$ takes fewer steps than executing $(PE\ e\ r\ c\ p\ h)$.

Base cases: Either e is a constant, or it is a variable. In the constant case, the specializer returns the value specification $(A\,e)$, which denotes at least e (under most type systems, exactly e). In the variable case, the specializer returns the value specification bound to the variable in the environment, which denotes all values which the variable could possibly evaluate to;*i.e.,* $r' \in (C\,(\textit{value-projection}\,r)) \Rightarrow (r'\,e) \in (C\,(\textit{value-projection}\,(r\,e)))$.

Inductive cases: The expression e is either a `LET`, `IF`, `PRIMCALL`, `USERCALL`, `LAMBDA`, or `ABSCALL`. We'll treat each case separately:

- `LET`: Each of the initialization expressions is evaluated, returning correct value specifications for each (by induction hypothesis). The environment is extended with these, which correctly approximates those values on which the body expression may be executed at runtime, so (by the hypothesis) evaluating the body in this environment also returns a correct value specification.

- `IF`: By the hypothesis, evaluating the predicate yields a correct approximation. If it's known, the appropriate arm is evaluated in the same environment, giving a correct result. If it's not known, then both arms are evaluated to correct approximations; by properties of the type lattice (*c.f.* Section A.1.1), the least upper bound of these approximations, which is returned, approximates all values that could be returned by either arm.

- `PRIMCALL`: By the hypothesis, all arguments are evaluated to correct approximations. The returned value specification is computed by executing the abstract primitive corresponding to the primitive, which (by the definition of an abstract primitive) produces a correct result. For example, consider our sample implementations. The code for `+` returns a concrete value only when both of its inputs are concrete numbers; otherwise it returns $\top_{Number}$, which is certainly safe (it merely says that `+` always returns a number). The code for `cons` always returns a pair containing the specifications of the arguments, while `car` either returns the specification of the `car` of the argument (if it's known to be a pair), otherwise $\top_{Val}$.

- `USERCALL`: By the hypothesis, all arguments are evaluated to correct approximations. The call is either unfolded or specialized. If it is unfolded, we evaluate the body of the call in an environment mapping the formal parameters to correct approximations to their values; by the hypothesis, this yields a correct result. If the call is specialized, the specification $\top_{Val}$ is returned, which is always safe.

- `LAMBDA`: This returns a value specification consisting of the text of the `lambda` form and an environment binding all of its free variables to (correct) approximations taken from the current environment. This denotes all possible closures that could be constructed from this code in a compatible environment, meaning that when we apply this closure (at an `ABSCALL`), the closed-over variables in the environment will be correctly initialized.

- `ABSCALL`: By the hypothesis, the head and arguments are evaluated to correct approximations. The call is either unfolded or left residual. If it is unfolded, then the head's value specification must be a closure; that closure's body (a source expression) is evaluated in an environment mapping the closure's free variables to the specifications in the closure's environment (assumed correct) and the bound variables to the specifications computed from the arguments (also assumed correct); thus, the evaluation of the body will produce a correct result. If the call is left residual, $\top_{Val}$ is returned.

■

Correctness of Residual Code

Now that we know the specializer computes correct approximations to runtime values, we must show that it uses these approximations to compute correct residual code. That is, executing the source and residual expressions in equivalent (modulo the differences between source and specialized environments described above) environments should yield identical results, provided that both environments are denoted by the specification.

There are two technical points here, involving restrictions on the environment and cache parameters to *PE*. First, any code expressions provided in the initial environment must be correct, in the sense that the residual code expressions of symbolic values in the environment must compute the same values as would be bound to those variables during the execution of the source program. This will not be a problem, since *PE-spec* only calls *PE* on newly instantiated environments (where all code expressions are merely references to formal parameter names), and *PE* only builds correct environment entries when evaluating `LET` expressions and performing unfolding.

Second, any specializations defined in the initial cache must be correct. That is, for the restricted set of actual parameters denoted by its argument specification (formals) each specialized function in the cache must compute the same function as the corresponding source function. This causes minor difficulties because, at intermediate stages during the construction of a specialization, its body may be undefined (*e.g.,* has the value *dummy*), which doesn't allow us to test this correctness property. Only when the specialization is complete is the body field of the corresponding *code-fcn* updated. Because the partial evaluator never examines the body of any *code-fcn*, but merely constructs references to the entire *code-fcn* object, the use of the dummy body doesn't affect the operation of the specializer, or its correctness, provided that a correct body is "filled in" before the residual program is executed.

We resolve this problem by noting that the cache starts out empty, and that "incomplete" specializations (*e.g.,* those with dummy bodies) are introduced only by *PE-spec*, which always fills them in later, so that the final cache (and thus the residual program) contains no references to incomplete specializations. Thus, the requirement we make is that each incomplete specialization in the cache must correspond to some invocation of *PE-spec* outside the current invocation of *PE*, and that the body eventually filled in by that invocation of *PE-spec* must be correct. We will denote this correspondence informally; for any incomplete *code-fcn g*, (*completion g*) will return the corresponding completed *code-fcn*,

while, for any complete *code-fcn*, it denotes the identity function. The existence of the outer invocation of *PE-spec* responsible for implementing *completion* will be left implicit, and its existence (and correctness) will be addressed in the portion of the proof dealing with `USERCALL` forms, rather than explicitly in the property statement.

Property 6: *Given program* $p \in Pgm$ *defining all function symbols in expression* $e \in Exp$, *specialization-time environment* $r \in Env$ *defined on all free variables in* e, *cache* $c \in Cache$, *and specialization-time environment* $h \in Env$ *defined on all free variables in* (*code-projection* r), *then for any Scheme environments* $r' \in (C\ (\textit{value-projection}\ r))$, $h' \in (C\ (\textit{value-projection}\ h))$, *and* $h'' = (\textit{spread-env}\ h\ h')$,

$$(\mathcal{E}'\ s\ h'') = (\mathcal{E}\ e\ r'\ p)$$
$$\textit{where}\ \langle s, c' \rangle = (PE\ e\ r\ c\ p\ h).$$

provided that

1. *for all free variables* x *in* e,

$$(\mathcal{E}'\ (r\ x)\ h'') = (r'\ x)$$

2. *for all function names* f *in* p *and argument specifications* a *of appropriate arity,* $(c\ f\ a)$ *is either undefined or returns a specialization* g *such that*

$$(\mathcal{E}'\ (\textit{code-fcn-body}\ (\textit{completion}\ g))\ (\textit{spread-env}\ t\ t')) = (\mathcal{E}\ (\textit{body}\ f)\ t'\ p)$$

where $t = [(\textit{code-fcn-formals}\ g)_i / (\textit{formals}\ f)_i]^n), t' \in (C\ (\textit{value-projection}\ t))$.

Proof: By induction on the number of calls to PE during the specialization of e.

Hypothesis: Property 6 is true for e', r', c', p, and h' where executing $(PE\ e'\ r'\ c'\ p\ h')$ takes fewer steps than executing $(PE\ e\ r\ c\ p\ h)$.

Base cases: If e is a constant, the property is trivially true. If e is a variable, it is bound to a symbolic value which was provided by the caller of PE, and whose value specification and code expression are explicitly assumed to be correct.

Inductive cases: The expression e is either a `LET`, `IF`, `PRIMCALL`, `USERCALL`, `LAMBDA`, or `ABSCALL`. We'll treat each case separately:

- `LET`: Each of the initialization expressions is evaluated, returning correct value specifications (by Property 5) and code expressions (by induction hypothesis) for each. The environment is extended with these, and thus both correctly approximates the values on which the body expression may be executed at runtime, and gives correct code expressions to compute those values at runtime. Thus, evaluating the body in this environment will return a correct code expression.

- `IF`: Evaluating the predicate yields a correct value specification (by Property 5) and a correct code expression. If the test is known; it's clearly safe to reduce the `IF`, so the appropriate arm is evaluated in the same environment, giving a correct result (by hypothesis). If the test is not known, then both arms are evaluated to correct approximations (by hypothesis); building a residual `IF` on correct code expressions for the test and arms is correct.

- `PRIMCALL`: By Property 5 and the hypothesis, all arguments are evaluated to correct value specifications and code expressions. The returned value specification is computed by executing the abstract primitive corresponding to the primitive, which (by the definition of an abstract primitive) produces a correct result. If the returned value specification is a ground value, then it's correct to return `no-code` as the code expression, since the value can be obtained with no computation at runtime. Otherwise, the code expression must compute the proper value. Consider consider our sample implementations. The primitive `+` returns `no-code` when both of its inputs are concrete numbers; otherwise, it returns an application of `+` to the specialized arguments (which are assumed to be correct by hypothesis). The primitive `cons` always returns a `cons` instruction; this isn't strictly necessary if both arguments are ground values, but it's certainly never wrong. The primitive `car` returns a `car` instruction if its argument is not known to be a pair. If the argument is known to be a pair, then the code for computing the value in its `car` slot is already present in the symbolic value in the `car` of the pair (and is assumed correct by hypothesis), so so `car` returns that symbolic value.

- `USERCALL`: By Property 5 and the hypothesis, all arguments are evaluated to correct value specifications and code expressions. The call is either unfolded or specialized. If it is unfolded, we evaluate the body of the call in an environment mapping the formal parameters to correct value specifications and code expressions; by the hypothesis,

this yields a correct result.

If the call is specialized, we know that the arguments on which specialization is performed are more general than the value specifications of the arguments (by Property 4). Depending on the cache, we have three cases:

- Specialization is undefined in cache. We extend the cache to define a new specialization with a dummy body, and construct its body by unfolding the body on value specifications equal to the (possibly abstracted) argument values (by Property 1 of instantiation), and code expressions which are references to formal parameters of the new specialization (by Property 2 of instantiation). Note also that, when we begin the specialization of the body, the environment and the "enclosing specialization environment" are equal. Thus, if we consider the recursive call to *PE* to compute the specialized body, we know (modulo spreading) $r' = r''$ and $(r\ x) = (\texttt{VAR}\ x)$, so $(\mathcal{E}'\,(r\,x)\,r'') = (r'\,x)$, and Restriction 1 holds. Restriction 2 holds because we know the new cache entry will be filled in with a correct body once the specialization is complete.[16] Thus, the hypothesis applies to the computation of the specialized body, and the specialized body is sufficiently general to be applicable in an environment where its formals are bound to the values passed by this call site, which is exactly what will happen at runtime.
- Specialization is defined in cache, and is complete. The specialization is correct by the property statement, and thus the generated residual call to that specialization is also correct.
- Specialization is defined in cache, but is incomplete. We return a residual call to the *code-fcn* object obtained from the cache. By the property statement, we know that this specialization is currently under construction by an enclosing invocation of *PE*, and will have a correct body when complete. Thus, it will be correct to invoke this *code-fcn* at runtime.

If the call is specialized, we return a residual call to a specialization whose body was

[16] This argument may appear to be circular, since we're using the hypothesis to show that the final body will be correct, allowing us to satisfy the hypothesis. This would normally require some sort of fixpointing argument (allowing us to satisfy a somewhat weaker hypothesis, and to iterate until we get the desired result). That is not the case here, because the constructed body is completely independent of the value of the body field of the new *code-fcn*; all that matters is that it reference the proper one, which we have "promised" to fill in later with the same specialization.

obtained by unfolding the body on equivalent or more general value specifications (by Property 4 of the termination mechanism, and Property 1 of instantiation), and code expressions which are references to formal parameters of the new specialization (by Property 2 of instantiation). Note also that, when we begin the specialization of the body, the environment and the "enclosing specialization environment" are equal. Thus, if we consider the recursive call to *PE* to compute the specialized body, we know (modulo spreading) $r' = r''$ and $(r\ x) = (\texttt{VAR}\ x)$, so $(\mathcal{E}'(r\,x)\,r'') = (r'\,x)$, and Restriction 1 holds. Restriction 2 holds because we know the new cache entry will be filled in with a correct body once the specialization is complete. Thus, the hypothesis applies to the computation of the specialized body, and the specialized body is sufficiently general to be applicable in an environment where its formals are bound to the values passed by this call site, which is exactly what will happen at runtime.

- `LAMBDA`: If the closure is not specialized, this constructs no residual code; all necessary information is present in the returned value specification. The symbolic values bound to variables in the environment of the returned closure are obtained from the current environment, and thus have correct value specifications and code expressions. If the closure is specialized, the specialization is obtained by unfolding the body of `lambda` in an environment mapping the formals to new symbolic values formed by instantiating $\top_{Val}$ on the formal parameter names. By the hypothesis and the properties of instantiation, this body will be sufficiently general to be applicable anywhere.

- `ABSCALL`: By Property 5 and the hypothesis, the head and arguments are evaluated to correct value specifications and code expressions. The call is either unfolded or left residual. If it is unfolded, then the body is executed in an environment mapping the closure's free variables to the symbolic values in the closure's environment (assumed correct) and the bound variables to the specifications computed from the arguments (also assumed correct); thus, the evaluation of the body will produce a correct result. If the call is left residual, applying a correctly specialized head expression to correctly specialized arguments is correct (just as in the `USERCALL` case above, Restriction 1 is satisfied because the arguments are freshly instantiated and $r = h$).

■

A.4 Fixpointing

In this section, we treat the fixpointing specialization algorithm of Chapter 4. As described in Chapter 4, this mechanism was not designed for general higher-order programs; we restrict ourselves to programs where all `ABSCALL` forms are unfoldable, and no `LAMBDA` forms need be specialized.[17] We begin by describing the changes made to the base specializer. We then proceed to define some useful properties of the specializer and its data structures, which then allow us to sketch an argument that the fixpointing algorithm generates correct code.

A.4.1 Changes to the Specializer

The main difference between the "base" specializer and the fixpointing specializer lies in the interpretation of the cache. In the base specializer, all residual calls to a specialization are assumed to be capable of returning any value whatsoever at runtime, so the specializer uses $\top_{Val}$ as the approximation to the value of such residual calls. This means that the body of a specialization computed by *PE-spec* does not depend on itself; the temporary presence of a dummy body in the *code-fcn* object for the specialization is undetectable. Thus, the sole purpose of the cache is to memoize the construction of the residual code for specializations.

The algorithm of Chapter 4 uses the cache for a second purpose: to record approximations to the return values of specializations. We choose to store each specialization's approximation in the value specification field of the symbolic value in the body field of the corresponding *code-fcn* object in the cache, and instantiate this approximation each time a residual call to the specialization is constructed. Computing and utilizing return value approximations requires several changes to the specializer, which are shown in Figures A.6 and A.7.

Utilizing return value approximations is simple; we merely alter the code in *PE* for constructing residual calls to specializations, returning an instantiation of the return value specification rather than one of $\top_{Val}$ (*c.f.* Figure A.6).

Computing appropriate return value approximations is more difficult. We must change the code in *PE-spec* for computing bodies of specializations. In base FUSE, none of the reductions performed during the computation of the body of a specialization depend upon any attribute of the specialization except the identity of its *code-fcn* object. This means

[17] This is not an onerous restriction, since similar-quality results for general higher-order programs can be obtained by CPS-converting the program and applying the analysis of Chapter 5 (*c.f.* Section A.5).

$$
\begin{array}{l}
PE\,[\![\,(\texttt{USERCALL}\ f\ e_i^n)\,]\!]\ env\ cache\ pgm\ senv \\
\quad = cond\ (unfold?\ f\ v_i^n\ cache) \rightarrow (PE\ b\ [s_i/x_i]\ cache''\ pgm\ senv) \\
\qquad\qquad where\ \langle g, cache''\rangle = PE\text{-}spec\ f\ v_i^n\ cache'\ Pgm \\
\qquad\qquad\quad where\ x_i^n = (formals\ f\ Pgm),\ b = (body\ f\ Pgm), \\
\qquad\quad else \rightarrow \langle (instantiate\ v\ (\texttt{USERCALL}\ f'\ s_i^n))\rangle,\ Cache''\rangle \\
\qquad\qquad\quad where\ v = (value\text{-}projection\ (code\text{-}fcn\text{-}body\ f')) \\
\qquad\qquad\qquad where\ \langle f', cache''\rangle = PE\text{-}spec\ f\ w_i^n\ cache'\ Pgm \\
\qquad\qquad\qquad\quad where\ w_i^n = (get\text{-}general\text{-}args\ f\ v_i^n) \\
\quad where\ v_i^n = (value\text{-}projection\ s_i)^n \\
\qquad where\ \langle s_i^n, cache'\rangle = PE\text{-}seq\ e_i^n\ env\ cache\ pgm\ senv
\end{array}
$$

Figure A.6: USERCALL code in *PE*, updated for return value approximations.

---

that no specialization need ever be recomputed.

Under our new interpretation of the cache, this is no longer the case. First, the body of a specialization *can* depend upon itself, because it may examine the return value approximation in its *code-fcn* object while it is being constructed, and use that value to perform reductions. Second, since one specialization can examine (and make use of) the return value approximations of other specializations, it may have to be recomputed if the approximations of those specializations are updated to more general values.

We address both of these differences in the revised version of *PE-spec* shown in Figure A.7. Because the body of a specialization can depend upon itself, we recompute it iteratively until it reaches a fixed point—this takes care of self-dependence. Dependence on other specializations is addressed by forcing all specializations constructed during a particular iteration to be recomputed on the next iteration—this is necessary because they might have made use of an overly specific return value for the current specialization, which is no longer valid.[18] However, it is safe to restart any recomputations on the existing return value approximations, so we don't want to remove the invalid cache entries completely. Thus, we add a boolean flag to each *code-fcn* object, signifying whether it is valid (*e.g.,* the return value approximation and code of the body can be used "as is") or invalid (*e.g.,* the body must recomputed). Only valid specializations are re-used, while invalid ones are recomputed. The helper function *invalidate-new-specs* takes two caches, c_1 and c_2, where c_2 is an

[18] FUSE avoids the need for this full "retraction" via a dependency tracking mechanism (*c.f.* Section 4.4.2).

```
PE-spec f v_i^n cache pgm
  = cond (code-fcn-valid? cache-result) → ⟨cache-result, cache⟩,
         else →
           cond (cache-result = ⊥_cache) → (compute-spec-body new-spec cache),
                    where new-spec = (LAMBDA s_i^n ⟨ ⊥_Val , no-code ⟩)
                (code-fcn-invalid? cache-result) → (compute-spec-body cache-result cache)
                  whererec compute-spec-body (spec cache) =
                                 begin
                                      (set-code-fcn-body! spec b)
                                      cond last = (value-projection b) → ⟨spec, cache''⟩,
                                           else → (compute-spec-body
                                                       spec
                                                       (invalidate-new-specs cache'' cache))
                                    where last = (value-projection (code-fcn-body spec))
                                          ⟨b, cache''⟩ =
                                           begin
                                              (set-code-fcn-valid! spec)
                                              (PE e env cache' pgm env)
                                                where
                                                    cache' = (update-cache cache f v_i^n spec)
           where env = [s_i/x_i]^n
                    where s_i^n = (instantiate v_i (VAR x_i))^n
                       where x_i^n = (formals f Pgm)
                             e = (body f Pgm)
      where cache-result = (cache f v_i^n)
```

Figure A.7: *PE-spec*, updated to compute return value approximations.

extension of c_1, and invalidates all specializations defined in c_2 but not in c_1.

As we shall see, the termination and correctness of this algorithm depend upon the return value increasing monotonically across iterations. This requires additional mechanism.

First, the cache must now obey a consistency requirement; we will call such caches "well-formed." Well-formedness will be defined formally in Section A.4.2; the basic idea is that specializations of a function on more general arguments must always have more general return values than specifications of the same function on more specific arguments. Otherwise, if respecializing the body under a more general return value assumption caused some call site to select a more general specialization, that site might then return a more specific return value, causing the body to return a more specific return value, violating monotonicity. We reestablish the well-formed-ness property each time we update the cache by using the helper function *update-cache* instead of performing a direct modification. (We will defer the code for *update-cache* until the partial ordering on caches has been defined; *c.f.* Section A.4.2).

Second, well-formed-ness is not enough to assure monotonicity. Because the return value of a specialization of f on a must approximate the return values of *any* execution of f on any argument vector denoted by a, the return value must also be more general than the value returned by any *unfolding* of f on $a' \sqsubseteq a$ (otherwise, choosing to unfold a call on one iteration of a fixed point loop, and choosing to specialize it on the next, might lead to nonmonotonicity if the specialization matched some incomplete specialization with a return value smaller than that of the unfolding). For the sake of simplicity, we will do the following. Before a call is unfolded, it is first specialized on the same arguments on which it would have been unfolded. This will compute a return value and update the cache appropriately. The call is then unfolded in that cache, returning the same value specification returned by the specialization (since the specialization is, by definition, a fixed point in the cache) and better code (since the overhead of calling the specialization and destructuring its return value has been eliminated).[19] The revised code is shown in Figure A.6.

[19] Of course, this is terribly inefficient. The implementation uses a more efficient solution; it prohibits unfolding of any call that might violate the well-formedness property (causing a cache entry to be created), but allows unfoldings that can be shown to have no effect on monotonicity. This mechanism interacts with the retraction mechanism in ways that we are not prepared to describe formally; suffice to say that it produces the same results as the solution presented here.

A.4.2 Basic Properties

Both the termination and correctness of this algorithm rest on a monotonicity property of the specializer. In this section, we first give some definitions relating to caches, allowing us to define monotonicity. We then show that the value specification and final cache returned by the specializer (*e.g.*, any call to *PE*) are monotonic functions of the environment and cache provided to the specializer.

Cache Orderings and Properties of Caches

A notion of monotonicity on caches requires a partial ordering on caches, so our first task is to define one. We will only ever compare a particular cache with other caches that extend it with additional specializations. This suggests the following definition: c_2 is larger (more general) than c_1 if and only if, for every specialization defined in c_1, c_2 defines the same specialization, with a more general return value. In other words, for all $c_1, c_2 \in \mathit{Cache}$, produced during specialization of a program $\mathit{pgm} \in \mathit{Pgm}$, $c_1 \sqsubseteq c_2$ iff

$$\forall \langle f, a \rangle \in (\mathit{Dom}\ c_1)\ [(c_1\ f\ a) \sqsubseteq (c_2\ f\ a)]$$

where the ordering on elements of *Code-fcn* is induced by the ordering on the value specifications of their bodies; *i.e.,*

$$\forall a, b \in \mathit{Code\text{-}fcn}$$
$$[a \sqsubseteq b \Leftrightarrow (\mathit{value\text{-}projection}\ (\mathit{code\text{-}fcn\text{-}body}\ a)) \sqsubseteq (\mathit{value\text{-}projection}\ (\mathit{code\text{-}fcn\text{-}body}\ b))]$$

We define a cache as *well-formed* if, for every specialization defined on a function f and argument specification a, the return value of that specialization is more general than the return values of any other specializations defined on f and a' where $a' \sqsubseteq a$. More formally, a cache c is well-formed iff for all $f \in \mathit{Fcn}$ and $a, a' \in \mathit{Val}^*$

$$\langle f, a \rangle \in (\mathit{Dom}\ f) \wedge \langle f, a' \rangle \in (\mathit{Dom}\ f) \wedge a' \sqsubseteq a \Rightarrow (c\ f\ a') \sqsubseteq (c\ f\ a)$$

The purpose of well-formed-ness is to ensure that caches properly reflect a monotonicity property of programs, namely that if executing a function f on some number of input vectors (elements of $(C\ a')$) yields some number of different return values, it must be the case that executing f on some larger set of input vectors (elements of $(C\ a)$) must yield all of the

same return values, and possibly more. All caches in FUSE are well-formed; we start out with an empty cache, and reestablish well-formedness each time we add a new specialization to the cache (in *PE-spec*) via the function *update-cache*, which takes a well-formed cache c, function f, argument specification a, and *code-fcn* g, and returns the smallest well-formed cache $c' \sqsupseteq c$ mapping $\langle f, a\rangle$ to g. The resulting cache has the same domain as $c\,[\langle f\ a\rangle \mapsto g]$, but may contain larger return value specifications for some specializations. Only the value specifications are altered; the code expressions are left unchanged. The following is a specification of *update-cache*.[20]

update-cache $c\ f\ a\ g =$
  the smallest well-formed cache $c'' \in$ *Cache such that*
    $c'' \sqsupseteq c'$,
    $(Dom\ c'') = (Dom\ c')$, *and*
    $\forall \langle f', a'\rangle \in (Dom\ c'')$
      $(code\text{-}projection\ (code\text{-}fcn\text{-}body\ (c''\ f'\ a'))) =$
      $(code\text{-}projection\ (code\text{-}fcn\text{-}body\ (c'\ f'\ a')))$
  where $c' = c\,[\langle f, a\rangle \mapsto g]$

Our algorithm relies on the following properties of *update-cache*:

Property 7: *The function update-cache always returns a cache larger than its cache argument. That is, for all well-formed caches* $c \in$ *Cache,* $f \in$ *Fcn,* $a \in$ *Val**, *and* $g \in$ *Code-fcn*

$$c \sqsubseteq (update\text{-}cache\ c\ f\ a\ g)$$

Property 8: *The function update-cache is monotonic in its cache and specialization arguments. That is, for all well-formed caches* $c, c' \in$ *Cache, functions* $f \in$ *Fcn, argument specifications* $a \in$ *Val**, *and code-fcn objects* $g, g' \in$ *Code-fcn*

$$c \sqsubseteq c' \wedge g \sqsubseteq g' \Rightarrow (update\text{-}cache\ c\ f\ a\ g) \sqsubseteq (update\text{-}cache\ c\ f\ a\ g')$$

Finally, we can further restrict the class of pairs of caches for which our monotonicity property must hold. First, since the cache is single-threaded through the partial evaluation,

[20] We give a specification rather than an implementation because there are several possible implementations. For example, FUSE performs the necessary "raising" of specializations lazily when cache lookup is performed, rather than eagerly when specializations are added.

we need only consider invocations of *PE* on caches c_1 and c_2 where c_2 *extends* c_1; that is, where c_2 is the result of some call to *PE* on c_1. This means that, for any specialization defined in c_2 but not in c_1, computing that specialization under c_1 must yield a more specific return value than that defined in c_2 (were this not the case, by the monotonicity of `update-cache`, and the transitivity of the partial ordering on caches, c_2 could not have been derived from c_1).

Second, the property need only hold for invocations of *PE* that are nested inside the same set of enclosing fixpoint loops. Thus, both caches will contain the same set of incomplete specializations (*e.g.,* valid specializations whose return values are still being computed); c_2 may contain some additional specializations, but all additional valid specializations in c_2 must be fixed points in c_2.

Monotonicity

Now that we have the terminology, we can state and prove the desired monotonicity property. For clarity, we define it as two interrelated sub-properties, which we prove in a single induction.

Property 9:

(a) For any fixed expression and program arguments, the function PE computes a monotonic function from the environment and cache arguments to the value specification and cache of the result. That is, for all $r_1, r_2 \in Env$ *and well-formed* $c_1, c_2 \in Cache$ *where* c_2 *extends* c_1*,*

$$r_1 \sqsubseteq r_2 \wedge c_1 \sqsubseteq c_2 \Rightarrow (\textit{value-projection}\ s_1) \sqsubseteq (\textit{value-projection}\ s_2) \wedge c_1' \sqsubseteq c_2'$$
$$\textit{where}\ \langle s_1, c_1' \rangle = (PE\ e\ r_1\ c_1\ p\ h_1),\ \langle s_2, c_2' \rangle = (PE\ e\ r_2\ c_2\ p\ h_2)$$

where $p \in Pgm$ *defines all function symbols in* $e \in Exp$*,* $h_1, h_2 \in Env$*, and* r_1 *and* r_2 *bind all free identifiers in* e*.*

(b) For any fixed function and program arguments, the function PE-spec computes a monotonic function from the argument specification and cache arguments to the value specification of body of the return specialization, and the returned cache. That is, for all $v_1, v_2 \in Val^*$ *and well-formed* $c_1, c_2 \in Cache$ *where* c_2 *extends* c_1*,*

$$a_1 \sqsubseteq a_2 \wedge c_1 \sqsubseteq c_2 \Rightarrow g_1 \sqsubseteq g_2 \wedge c_1' \sqsubseteq c_2'$$
$$\textit{where } \langle g_1, c_1' \rangle = (\textit{PE-spec } f\ a_1\ c_1\ p),\ \langle g_2, c_2' \rangle = (\textit{PE-spec } f\ a_2\ c_2\ p)$$

Proof: By induction on the number of calls to *PE* during the execution of *PE* or *PE-spec*.

Hypothesis: Property 9(a) is true for e', r_1', r_2', c_1', c_2', p, h_1', and h_2' where executing $(PE\ e'\ r_1'\ c_1'\ p\ h_1')$ takes fewer steps (calls to *PE*) than executing $(PE\ e\ r_1\ c_1\ p\ h_1)$, and executing $(PE\ e'\ r_2'\ c_2'\ p\ h_2')$ takes fewer steps than executing $(PE\ e'\ r_2\ c_2\ p\ h_2)$. Property 9(b) is true for for f', a_1', a_2', c_1', c_2', and p, where executing $(\textit{PE-spec } f'\ a_1'\ c_1'\ p)$ takes fewer steps (calls to *PE*) than executing $(\textit{PE-spec } f\ a_1\ c_1\ p)$ and $(\textit{PE-spec } f'\ a_2'\ c_2'\ p)$ takes fewer steps than executing $(\textit{PE-spec } f\ a_2\ c_2\ p)$.

We will separate our arguments for parts (a) and (b). We begin with part (a):

Base cases: Trivially true for constants. For any variable x, $r \sqsubseteq r' \Rightarrow (r\,x) \sqsubseteq (r'\,x)$.

Inductive cases: The expression e is either a `LET`, `IF`, `PRIMCALL`, `USERCALL`, `LAMBDA`, or `ABSCALL`.

Note that specializing a sequence of argument forms using *PE-spec* single-threads the cache, so that (by transitivity of the ordering on caches) the hypothesis is applicable to each sub-specialization.

- `LET`: Evaluating the initializers preserves the necessary relation on caches, and the environment used for specializing the body is more general under r_2 than under r_1 (by hypothesis), so the hypothesis also applies to the evaluation of the body.
- `IF`: True by hypothesis for the test and arms. Let the value specifications returned for the predicate, consequent, and alternative be p_1, x_1, y_1, and p_2, x_2, and y_2 under r_1, c_1 and r_2, c_2, respectively. We see the cases:
  - Conditional is reducible for both p_1 and p_2. It must reduce to the same arm on both cases. Since $c_1 \sqsubseteq c_2$ and $x_1 \sqsubseteq x_2$, the property is true.
  - Conditional is irreducible for both p_1 and p_2. Then the returned value specification is $x_1 \sqcup y_1$ under r_1 and $x_2 \sqcup y_2$ under r_2. Thus $x_1 \sqcup y_1 \sqsubseteq x_2 \sqcup y_2$, and the property is true.
  - Conditional is reducible for p_1 but not p_2. Suppose p_1 is true. Then we have $x_1 \sqsubseteq x_2 \sqcup y_2$. Similarly, for p false, $y_1 \sqsubseteq x_2 \sqcup y_2$.

– Conditional is reducible for p_2 but not p_1. Not possible, since we know $p_1 \sqsubseteq p_2$.

The necessary relation on the cache is preserved in all cases by the hypothesis and single-threading.

- `PRIMCALL`: True by hypothesis for the arguments. True for the return value by monotonicity of abstract primitive operators (*c.f.* Section A.1.2). True for the return cache because applying the primitive doesn't affect the cache.

- `USERCALL`: True by hypothesis for the arguments. We analyze the following cases:

  – Specialize under r_1, c_1, specialize under r_2, c_2. We know that the value specifications a_1 and a_2 passed to *PE-spec* have the relation $a_1 \sqsubseteq a_2$ (by hypothesis and monotonicity of *get-general-args*); we also know that the caches c_1' and c_2' passed to *PE-spec* satisfy $c_1' \sqsubseteq c_2'$ (by hypothesis). The value specification of the returned symbolic value is taken from the body of the specialization returned by *PE-spec*, and returned cache is that returned by *PE-spec*. All calls to *PE* in *PE-spec* involve fewer steps than this call. Thus, the property is true by hypothesis(b).

  – Unfold under r_1, c_1, specialize under r_2, c_2.

  Remember that we first specialize to obtain a new cache, then unfold. The property is true for the the calls to *PE-spec* under both r_1, c_1 and r_1, c_2 by the argument above. Because the specialization returned by *PE-spec* under r_1, c_1 is a fixed point in the cache returned by *PE-spec*, the unfolding will return the same return value and cache, just a different code expression. Thus, the property holds.

  – Specialize under r_1, c_1, unfold under r_2, c_2. This can't happen, as it would violate the monotonicity of the function *unfold?* (Property 3).

  – Same argument as the unfold/specialize case.

- `LAMBDA`: The closure is not specialized under either context. The property is trivially true; the returned closures inherit the ordering of the environments they capture, and the cache isn't altered.

- `ABSCALL`: The call is always unfolded (*i.e.*, the head is always known). The unfolding is performed under environments r_1', r_2' contained in the closure being applied, which

obey $r_1' \sqsubseteq r_2'$ by hypothesis. Thus, the hypothesis applies to the unfolding, and the property is true.

Now, we turn to part (b).

Base cases: Both specializations are found in the cache, and are valid. We know $(c_1\ f\ v_1) \sqsubseteq (c_2\ f\ v_2)$ by the ordering on caches. Thus, the property is true for return values. Since the caches aren't updated, the property is true for caches also.

Inductive cases: At least one of the caches does not define the specialization, or defines an invalid specialization (requiring respecialization). This give the cases:

- Specialization must be computed in both c_1 and c_2. Consider the two fixpoint loops, running in parallel (until both converge; extra iterations on the faster-converging loop won't hurt). The caches d_1, d_2 just before we call PE on the body are c_1, c_2 extended with single entries mapping the new specialization to a *code-fcn* with some return value a_1, a_2 (where $a_1 \sqsubseteq a_2$, due to the well-formed-ness of d_1 and d_2 as enforced by *update-cache*), so $d_1 \sqsubseteq d_2$. By hypothesis (a), the return value approximations after this iteration will be a_1', a_2' where $a_1' \sqsubseteq a_2'$; also, the returned caches d_1', d_2 will satisfy $d_1' \sqsubseteq d_2'$. Thus, if this was the final iteration, the property holds. If it wasn't the final iteration, we recompute d_1, d_2 (on the larger a_1, a_2), and all of the same conditions hold for the next iteration.

- Specialization must be computed in c_1, but not c_2. By the assumption that c_1 and c_2 define the same set of incomplete specializations, c_2 must define a valid specialization that is also a fixed point in c_2. If we recompute that specialization, we'll get the same answer, and the argument above applies.

- Specialization must be computed in c_2, but not c_1. In this case, c_1 must define a valid specialization, and c_2 an invalid one (c_1 must contain the specialization, and since c_2 extends c_1, c_2 must also). But the valid specialization is a fixed point in c_1, so just consider recomputing both specializations.

■

Termination

Given this monotonicity result, we can show that partial evaluation terminates. Termination of the fixpoint iteration algorithm requires two things: (1) performing a finite number of

iterations, and (2) performing a finite amount of work on each iteration. The latter property is assured by the base FUSE termination mechanism, which must guarantee that specializing any expression in any context (in this case, the body of the procedure whose return value is being computed, in the environment binding its free variables appropriately) returns a finite residual code expression. The former property is not an issue in base FUSE, and must be explicitly treated here.

The fixpoint iteration process halts when the same return value approximation is computed on two successive iterations. This will always happens (eventually) because (by Property 9) the computed return values rise monotonically in the type lattice on each iteration, and there are no infinite ascending chains in the lattice.

A.4.3 Correctness

Now we have the property we want, namely that fixpoint iteration will find a least fixed point in a finite number of iterations. As in base FUSE, we will show the correctness of our algorithm in two parts. First, we will show that, at each step during the specialization process, the specializer computes correct approximations to the sets of values that could be returned by source code expressions executed under the given approximations to the values of their free variables, and to the return values of procedure calls. Then, we will show that, given such correct approximations, the specializer will produce sufficiently general residual code.

Correctness of Type Inference

The type correctness property for the fixpointing specializer is is similar to Property 5, except that now the correctness of the result must be conditionalized on the correctness of the return value approximations in the initial cache. At first, this might seem simple, since we need only compare the results of evaluating the source function and the specialization in all environments denoted by the specialization's argument approximations. This would be the case if all specializations in the cache were complete, but due to the iterative nature of the computation of return value approximations, specializations must be permitted to have incorrect (overly specific) return value approximations during specialization, provided that the final approximations (used to construct the residual program) are correct. We will see that specifying the dependence of the approximations on the initial cache will be somewhat

complex, so we will treat it only informally.[21]

First, consider specializing a single self-recursive function f where all recursive calls in f's body happen to have arguments matching the specialization being constructed (this is just a monovariant fixpointing return type analysis on f). The standard correctness argument goes something like the following. The initial return value approximation $v_0 = \perp_{Val}$ may be incorrect; thus, after f's body is analyzed under this approximation, the new approximation v_1 is correct for all executions of f making no recursive calls to f (provided, of course, that the type evaluator computes correct results for primitives, conditionals, etc). Iteratively re-analyzing f's body under each v_i yields a v_{i+1} that is correct for all executions of f with recursion depth less than or equal to i. Provided that the type evaluation function is monotonic, we will eventually find a fixed point v_k such that $v_k = v_{k+1}$, which is correct for all executions of f, regardless of the recursion depth, and the argument is complete. This argument can be extended to the monovariant specialization of programs containing an arbitrary number of functions, by maintaining a separate approximation v_i^f for all functions f in the program, and demonstrating that depth-first specialization with invalidation monotonically raises all of the v_i^f.

In a polyvariant specializer, we can have many different specializations of the *same* source function, each with its own return value approximation. This complicates things, because now the restrictions on recursion depth apply to particular specializations of source functions, rather than to the source functions themselves. We assign each specialization a return value approximation $v_i^{(f,a)}$ where f is the name of the source function, a is the argument vector, and i is the maximum recursion depth (in terms of calls to the specialization (f, a)) for which the approximation is valid. Of course, the correctness of the approximation is also limited by the correctness of any approximations to the return values of *other* specializations it invokes. We can make this explicit by having the specializer maintain a *stack* of pairs $\langle g, i \rangle$, one for each specialization g currently being constructed, giving the maximum recursion depth (in terms of calls to g) for which that specialization's return value (*e.g.*, (*value-projection* (*code-fcn-body* g)) is correct. To avoid having to add yet another parameter to *PE*, we make the stack explicit in the cache, by having each specialization g of f contain a copy of the stack in effect at the beginning of the current evaluation of f's

[21] Treating this properly would require that we define an "instrumented" concrete semantics for source programs that computed (*argument value*, *return value*) pairs for all function applications, and then demonstrate that the partial evaluator computes a conservative, monotonic abstraction of the instrumented semantics. For an example of a proof in this style, see [105].

body.

Now, we can apply an argument similar to the one given above for monovariant specialization, and deduce that, when a fixed point is reached, the return value approximation for that specialization is correct for all executions of that specialization on arguments denoted by the specialization's argument vector, and obeying the constraints on other specializations expressed in its stack. However, we're interested in the approximation being correct for corresponding executions of the *source* function that was specialized (we don't want to address the correctness of the specialized code until after we've proven the type correctness property). To make this connection, consider modifying the specializer not to perform delta-reductions. That is, for each symbolic value returned by *PE*, the value specifications are computed exactly as before, but any primitives and conditionals are left residual in the code expressions (unfolding and specialization of `USERCALL` forms is still allowed; `ABSCALL` forms are unfolded to obtain a return value, but a residual `ABSCALL` is still constructed). Since only delta-reductions reduce the domain of the specialization, every specialization in the residual program has exactly the same meaning as the corresponding source function; the programs are equivalent, modulo the translation from Scheme to graphical form, and some amount of duplication due to unfolding and specialization.[22] Thus, proving the correctness of a return value approximation w.r.t. the execution of a specialization is the same as proving it w.r.t the execution of the source function.[23]

We will state our property in terms of this modified specializer (let's call it $\widehat{PE}$). Since $\widehat{PE}$ produces exactly the same type approximations as *PE*, it should be clear that the desired property also applies to *PE*, provided that, given correct type approximations, *PE* generates correct code (*c.f.* Property 11).

Property 10: *For any expression $e \in Exp$ which is part of a program $p \in Pgm$, and for any specialization-time environment $r \in Env$ defined on all free variables in e, cache $c \in Cache$, and "enclosing specialization environment" $h \in Env$, the value specification (value-projection s) of the symbolic value s returned by specializing e on r, c, p, and h*

[22] This is akin to performing inlining and procedure cloning [31], where duplication is performed in the hope that it will enable delta-reduction optimizations (constant folding/propagation, dead code elimination, etc) in some later optimization pass.

[23] This complication is due to the fact that the specializer performs type inference and constructs residual code simultaneously. Not only is this inefficient (since all code constructed on the first $k-1$ iterations of a fixed point loop gets discarded), but it adds conceptual complexity. We now believe that performing a polyvariant type analysis, followed by a separate code construction pass [70], is a cleaner method.

denotes all values which could be returned by executing e *under any Scheme environment denoted by* r*;* i.e., *for all Scheme environments* $r' \in (C\,(\textit{value-projection}\,r))$

$$(\mathcal{E}\,e\,r'\,p) \in (C\,(\textit{value-projection}\,s))$$
$$\textit{where } \langle s, c'\rangle = (\widehat{PE}\,e\,r\,c\,p\,h).$$

provided that for every function name f *in* p *and argument specification* a *of appropriate arity, every valid*[24] *specialization* $g = (c\ f\ a)$ *in the cache has a correct return value approximation;* i.e.,

$$(\mathcal{E}\,(\textit{body}\,f)\,t'\,p) \in (\textit{value-projection}\,(\textit{code-fcn-body}\,g))$$

for all $t' \in (C\,(\textit{value-projection}\,t))$, $t = [(\textit{code-fcn-formals}\,g)_i/(\textit{formals}\,f)_i]^n$, *provided that, for all entries* $\langle g', i\rangle \in (\textit{stack}\,g)$, *executing*

$$(\mathcal{E}'\,(\textit{code-fcn-body}\,(\textit{completion}\,g))\,(\textit{spread-env}\,t\,t'))$$

does not exceed a recursion depth of i *for specialization* g'.

Proof Sketch: By induction on the number of calls to $\widehat{PE}$ during the specialization of e. This is very similar to the proof of Property 5; the only interesting difference is in the inductive case on `USERCALL` forms, for the subcase where the call is specialized.[25]

If the cache defines the specialization, the property is true by hypothesis. If the cache does not define the specialization, we must show that the value obtained by fixpoint iteration is correct (given the assumptions about the return values of incomplete specifications in the current cache). We know (by hypothesis) that the approximation returned by each iteration is correct for the environment and cache in which it was constructed, and for one more level of recursion (in terms of the current specialization) than the prior approximation; thus, the fixed point is correct the environment and cache in which it was constructed, and an arbitrary recursion depth. By monotonicity (Property 9), we know that a fixed point will be found, and thus the property is true.

■

[24] Invalid specializations don't matter, since they get rebuilt—their sole purpose is to guarantee monotonicity of the cache in Property 9.

[25] In the proof of Property 5, this was trivially true because all residual calls returned a value specification of $\top_{Val}$.

Correctness of Residual Code

Given that the value specifications are correct, the correctness of the specialized expression follows easily. The decision of how to reduce or residualize an expression depends solely on the value specifications computed for the expression's subforms, and does not change when we add fixpointing. That is, the value specifications computed for subforms may be more accurate, but the use of that information remains the same.

Property 11: *Given program $p \in Pgm$ defining all function symbols in expression $e \in Exp$, specialization-time environment $r \in Env$ defined on all free variables in e, cache $c \in Cache$, and specialization-time environment $h \in Env$ defined on all free variables in (code-projection r), then for any Scheme environments $r' \in (C\ (\textit{value-projection}\ r))$, $h' \in (C\ (\textit{value-projection}\ h))$, and $h'' = (\textit{spread-env}\ h\ h')$,*

$$(\mathcal{E}'\ s\ r'') = (\mathcal{E}\ e\ r'\ p)$$
$$\textit{where } \langle s, c' \rangle = (PE\ e\ r\ c\ p\ h).$$

provided that

1. *for all free variables x in e,*

$$(\mathcal{E}'\ (r\ x)\ h'') = (r'\ x)$$

2. *for all function names f in p and argument specifications a of appropriate arity, every valid specialization $g = (c\ f\ a)$ in the cache satisfies*

$$(\mathcal{E}\ (\textit{body}\ f)\ t'\ p) \in (\textit{value-projection}\ (\textit{code-fcn-body}\ g))$$

and

$$(\mathcal{E}'\ (\textit{code-fcn-body}\ (\textit{completion}\ g))\ (\textit{spread-env}\ t\ t')) = (\mathcal{E}\ (\textit{body}\ f)\ t'\ p)$$

for all $t' \in (C\ (\textit{value-projection}\ t))$, $t = [(\textit{code-fcn-formals}\ g)_i/(\textit{formals}\ f)_i]^n$, provided that, for all entries $\langle g', i \rangle \in (\textit{stack}\ g)$, executing

$$(\mathcal{E}'\ (\textit{code-fcn-body}\ (\textit{completion}\ g))\ (\textit{spread-env}\ t\ t'))$$

does not exceed a recursion depth of i for specialization g'.

(This is essentially the same as Property 6, with the added restriction that the initial cache contain correct return value approximations for specializations).

Proof Sketch: By induction on the number of calls to *PE* during the specialization of e. This is very similar to the proof of Property 6, so we won't go through it in detail here. The only code expressions that make it into the final residual program are those constructed during the final iteration of all active fixed point loops, at which point all return value approximations are correct by Property 10. Given that, the correctness the residual code expressions produced by individual reductions follows, as these are unchanged from the code in the base specializer.

■

A.5 CFA

In this section, we treat the control-flow-analysis-based iterative specialization algorithm of Chapter 5. We describe the changes made to the specializer, some basic monotonicity and termination properties, and give a proof sketch of a correctness property.

A.5.1 Changes to the Specializer

Basics

First, we treat only CPS-converted source programs.[26]. By "CPS-conversion," we do not mean the full CPS transformation of [108]; we require only that all function definitions (*i.e.,* top-level definitions and `LAMBDA` forms not representing continuations) accept continuation parameters, and that `IF` expressions be CPS-converted. Expressions consisting of primitive calls are permitted, as this saves us from having to rewrite the existing specializer's code for primitives. The `LET` form is removed from the input language.

Second, we assume a function *SITES* which, given a symbolic value p denoting a residual program, and symbolic value c whose value specification is a closure, returns a set of symbolic values $(SITES\ p\ c)$ such that for any runtime closure constructed by $(\textit{code-projection}\ c)$, all of its textual call sites in the residual program are contained in $(\textit{code-projection}\ (SITES\ p\ c)_i)^n$.

[26] This is not essential for the algorithm, but simplifies both its implementation and analysis

This is just a standard control flow analysis function as in [105, 50, 104]; for readers seeking formal definitions, we suggest [105]. We require that control flow analysis be correct in the usual sense (*i.e.,* it only overestimates, never underestimates, the set of call sites).

Third, we change the way closures are specialized. In base FUSE, we simply specialize all closures needing to be specialized[27] on completely unknown arguments (*e.g.,* an appropriate-arity vector of $\top_{Val}$). Under the iterative technique, we instead use the *SITES* function to compute (an approximation to) the closure's set of call sites, and specialize the closure on an argument approximation that is the least upper bound of the old approximation and the argument approximations at all of the call sites. Initially, when a closure is constructed, its set of call sites will be empty, but as the program grows, so may the set of call sites. This may force respecialization of the closure, which may affect the call sites of both this closure and other closures, thus motivating the iterative algorithm of Section 5.3.

Changes

Most of the specializer is left unchanged; we need only change the code for constructing closures, and add code to specialize them. We begin with the `LAMBDA` special form. The returned symbolic value has a value specification consisting of the closure, and a code expression consisting of the closure specialized on no call sites (*e.g.,* the formal parameters are $\perp_{Val}$ and the body is empty).

$$
\begin{aligned}
&PE[\![\texttt{(LAMBDA } x_i^n \ e\texttt{)}]\!]\ env\ cache\ pgm\ senv \\
&\quad = \langle (instantiate\ \langle x_i^n, e, env\rangle,\ spec\text{-}code),\ cache\rangle \\
&\qquad where\ spec\text{-}code = \texttt{(LAMBDA } (instantiate\ \perp_{Val}\ \texttt{(VAR } x_i\texttt{))}^n\ dummy\texttt{)}
\end{aligned}
$$

The other major change comes in the function *PE-program*. After the specialization of the entire program is complete, we may find that the call site sets of various closures in the specialized program may have changed, necessitating respecialization. To detect this, we walk the residual program to find all closure-valued symbolic values, using the function *find-closures*,[28] and use the control flow analysis function *SITES* to find each closure's set of

[27] Recall that we assumed (and implemented; *c.f.* Section 3.3.5) a simple oracle which tells us when a closure will only be unfolded, and thus will not need to be specialized.

[28] The function *find-closures* walks a graph of symbolic values and returns all symbolic values whose value specifications are closures. The actual implementation performs this incrementally during specialization, and never needs to walk the residual program graph outside of the code generator. Thus, we will not show the implementation of *find-closures*.

call sites. Given the set of call sites, we compute a new argument vector by taking the least upper bound of the existing argument vector and the argument specifications at each call site. If the argument vector has changed, we queue the closure for respecialization. Once all closures have been examined, if no respecialization is necessary, the residual program is complete; otherwise, we perform any required respecializations and repeat the analysis loop. The revised code for *PE-program* is shown in Figure A.8.

A.5.2 Basic Properties

Monotonicity

We first note that argument specifications computed for any particular closure rise monotonically during specialization.

Property 12: *During the execution of PE-program on any program $p \in Pgm$, goal function $f \in Fcn$, and argument specification $v_i^n \in Val^*$, for any closure-valued symbolic value s processed by walk-closures, the new argument vector args′ and the old argument vector args always have the relationship $args \sqsubseteq args'$. (This is trivially true since args' is defined to be the least upper bound of args and the argument vectors obtained by control flow analysis).*

Termination

Given this, we can show that the specialization algorithm terminates. To terminate, the iterative specialization algorithm must:

1. Construct a finite number of closures, and

2. Specialize each closure a finite number of times

Issue (1) is addressed by the base FUSE termination mechanism, which assures that every top-level procedure is unfolded only a finite number of times, and that only a finite number of specializations of each top-level procedure are constructed. Closures are constructed in two ways: by executing `lambda` expressions and by instantiating closure-valued value specifications when procedures are specialized on actual parameters containing closures. Since closures are constructed by evaluating `lambda` expressions at specialization time, and a `lambda` expression is evaluated at most as many times as its enclosing top-level

$$
\begin{array}{l}
\textit{PE-program pgm goal } v_i^n \\
\quad = \textit{begin} \\
\qquad\quad (\textit{closure-loop } (\textit{code-fcn-body } f)\ \textit{cache}') \\
\qquad\quad f \\
\qquad \textit{where } \langle f, \textit{cache}' \rangle = (\textit{PE-spec goal } v_i^n\ []\ \textit{pgm}) \\
\\
\textit{closure-loop } g\ \textit{cache} \\
\quad = \textit{cond } (\textit{null? } r) \rightarrow \textit{cache}, \\
\qquad\quad \textit{else} \rightarrow (\textit{closure-loop } g\ (\textit{respecialize-closures } r\ \textit{cache})) \\
\qquad \textit{where } r = (\textit{walk-closures } g\ (\textit{find-closures } g)) \\
\\
\textit{respecialize-closures } r\ \textit{cache} \\
\quad = \textit{cond } (\textit{null? } r) \rightarrow \textit{cache}, \\
\qquad\quad \textit{else} \rightarrow \textit{begin} \\
\qquad\qquad\qquad (\textit{set-lambda-formals! } l\ s_i^n) \\
\qquad\qquad\qquad (\textit{set-lambda-body! } l\ b) \\
\qquad\qquad\qquad (\textit{respecialize-closures } (\textit{rest } r)\ \textit{cache}') \\
\qquad\qquad\quad \textit{where } \langle b, \textit{cache}' \rangle = \textit{PE } e\ \textit{env } [s_i/x_i]^n\ \textit{cache pgm} \\
\qquad\qquad\qquad \textit{where } s_i^n = (\textit{instantiate args}_i\ \texttt{(VAR } x_i\texttt{)})^n \\
\qquad\qquad\qquad\quad \textit{where } \langle \langle x_i^n, e, \textit{env} \rangle, l \rangle = \textit{sval} \\
\qquad\qquad\qquad\qquad \textit{where } \langle \textit{sval}, \textit{args} \rangle = (\textit{first } r) \\
\\
\textit{walk-closures } (g\ c\ r) \\
\quad = \textit{cond } (\textit{null? } c) \rightarrow \textit{nil} \\
\qquad\quad \textit{else} \rightarrow \textit{cond } (= \ \textit{args args}') \rightarrow r, \\
\qquad\qquad\qquad \textit{else} \rightarrow \langle \langle s, \textit{args}' \rangle, r \rangle \\
\qquad\qquad\quad \textit{where args}' = \textit{args} \sqcup (\bigsqcup \textit{site-args}_i^n) \\
\qquad\qquad\qquad \textit{where site-args}_i^n = (\textit{value-projection} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\textit{usercall-args } (\textit{SITES } f\ c)_i))^n, \\
\qquad\qquad\qquad\qquad \textit{args} = (\textit{value-projection} \\
\qquad\qquad\qquad\qquad\qquad\quad (\textit{lambda-formals } (\textit{sval-code } s))) \\
\qquad\qquad\qquad\quad \textit{where } s = (\textit{first } c)
\end{array}
$$

Figure A.8: *PE-program* and helper functions for iterative specialization of closures

procedure is evaluated[29], a finite number of closures are constructed from `lambda` expressions. Since a finite number of specializations are constructed, any closure constructed from a `lambda` expression can be duplicated by instantiation at most a finite number of times.

Issue (2) is addressed by (1) and Property 12, giving:

Property 13: *During the execution of PE-program on any program $p \in Pgm$, goal function $f \in Fcn$, and argument specification $v_i^n \in Val^*$, the function respecialize-closures processes at most a finite number of pairs $\langle sval, args \rangle$ where $sval \in Sval$ and $args \in Val^*$.*

Proof: Property 13 is true because

- There is only one closure-valued symbolic value *sval* per closure,
- The argument vectors *args* for each closure rise monotonically (Property 12), and
- The lattice contains no infinite ascending chains.

■

Thus, the iterative algorithm always terminates.

A.5.3 Correctness

As in base FUSE, we will show the correctness of our algorithm in two parts: (1) correctness of the value specifications returned by specializing source forms with respect to specialization-time environment (*i.e.,* the value specifications must correctly approximate the set of values returnable by the source expression at runtime), and (2) correctness of the residual code (*i.e.,* executing the source and residual expressions in equivalent (modulo the differences between source and specialized environments) environments should yield identical results, provided that both environments are denoted by the specification.

[29] Recall that we allow nesting of `lambda` expressions in source programs, but require that all loops pass through top-level procedures (*c.f.* Section 3.1.1). This, of course, still requires recursion detection across closure invocation, which is described in Section 3.3.5.

Type Correctness

Property 14: *For any expression $e \in Exp$ which is part of a CPS-converted source program $p \in Pgm$, and for any specialization-time environment $r \in Env$ defined on all free variables in e, cache $c \in Cache$, and "enclosing specialization environment" $h \in Env$, the value specification (value-projection s) of the symbolic value s returned by specializing e on r, c, p, and h denotes all values which could be returned by executing e under any Scheme environment denoted by r;* i.e., *for all Scheme environments $r' \in (C\,(\textit{value-projection}\,r))$*

$$(\mathcal{E}\,e\,r'\,p) \in (C\,(\textit{value-projection}\,s))$$
$$\textit{where } \langle s, c' \rangle = (PE\,e\,r\,c\,p\,h).$$

Proof Sketch: Because the source program is CPS-converted, the only forms that return values are `CONST`, `VAR`, `PRIMCALL`, and `LAMBDA`.[30] The specializer code for computing the return values of these forms is unchanged from base FUSE. Thus, the proof is a degenerate form of the proof of Property 5; the arguments for `CONST`, `VAR`, `PRIMCALL`, and `LAMBDA` are unchanged.

■

Residual Code Correctness

Property 15: *Given any expression $e \in Exp$ which is part of a program $p \in Pgm$, specialization-time environment $r \in Env$ defined on all free variables in e, cache $c \in Cache$, and specialization-time environment $h \in Env$ defined on all free variables in (code-projection r), then for any Scheme environments $r' \in (C\ (\textit{value-projection}\ r))$, $h' \in (C\ (\textit{value-projection}\ h))$, and $h'' = (\textit{spread-env}\ h\ h')$,*

$$(\mathcal{E}'\,s'\,h'') = (\mathcal{E}\,e\,r'\,p)$$
$$\textit{where } s = \textit{begin}$$
$$(\textit{closure-loop}\,s\,c')$$
$$s$$
$$\textit{where } \langle s, c' \rangle = (PE\,e\,r\,c\,p\,h).$$

[30] The forms `IF`, `USERCALL`, and `ABSCALL` never return (since the arms of conditionals and the functions invoked by calls are CPS-converted), while the `LET` form is removed from the input language.

provided that

1. *for all free variables x in e,*

$$(\mathcal{E}'\,(r\,x)\,h'') = (r'\,x)$$

2. *for all function names f in p and argument specifications a of appropriate arity, $(c\,f\,a)$ is either undefined or returns a specialization g such that*

$$(\mathcal{E}'\,(\textit{code-fcn-body}\;(\textit{completion}\;g))\,(\textit{spread-env}\;t\;t')) = (\mathcal{E}\;(\textit{body}\;f)\;t'\;p)$$

where $t = [((\textit{code}-\textit{fcn}-\textit{formals})\;g)_i/(\textit{formals}\;f)_i]^n), t' \in (C\;(\textit{value-projection}\;t))$.

Proof Sketch: First, we must show that the code generated by *PE* is correct, provided that all `ABSCALL` forms in the residual programs invoke sufficiently general specialized closures. With the exception of the code for specializing closures, the specializer is unchanged from that of base FUSE, so the proof of Property 6 applies. The only difference is that we can no longer assume that the code expressions returned by `LAMBDA` forms are sufficiently general to be applicable anywhere.

Given that, we can use induction and the monotonicity property for argument approximations (Property 12) to show that sufficiently general residual `LAMBDA` forms are generated. The basic idea is as follows:

After the kth invocation of *closure-loop*, the residual `LAMBDA` expressions in the residual program g are general enough to be applicable at all call sites found in the $(k-1)$th iteration of *closure-loop* (consider the "0th" iteration as finding no call sites). The per-closure argument approximations computed on the final (nth) iteration are a fixed point of *closure-loop*, meaning that a $(n+1)$th iteration would compute the same residual program (since the residual code is a deterministic function of the argument approximations). Since the residual `LAMBDA` expressions in the residual program computed by the $(n+1)$th iteration are general enough for all applications in the nth residual program, and the nth and $(n+1)$th residual programs are the same, the specializations computed by the nth iteration are general enough to be applicable at all call sites in the nth program, and the proof is complete.

■

A.6 Re-use

In this section, we treat the re-use analysis of Chapter 6. We will consider it only in the context of base FUSE, and will not show that it also works if fixpointing or CFA are used (the necessary changes, and informal arguments about their correctness, are given in Section 6.4). Also, the "base" mechanism of Chapter 6 does not work for general higher-order programs without modification; we restrict ourselves to programs where all `ABSCALL` forms are unfoldable, and no `LAMBDA` forms need be specialized. Section 6.4.2 proposes several modifications to overcome this restriction; we will not treat them here.

As we described in Chapter 6, the goal in re-use analysis is to compute, for each specialization of a source program function f, a safe approximation to the set of argument values for which the specialization computes the same result f. As this set is often larger than the set of values denoted by the argument specification on which the specialization was constructed ("cache index," in the terminology of Chapter 6), we can use it to safely re-use specializations of f in a larger number of contexts. We call this new approximation the MGI, or *Most General Index*, of the specialization. In this section, we will describe a method for computing the MGI, and will show that the MGIs computed by this method (1) are correct MGIs, and (2) have certain monotonicity properties that allow them to be used in determining not only when it is *safe* to re-use a specialization, but also when it is *optimal* to do so.

A.6.1 Changes to the Specializer

The changes to the specializer fall into three categories: representing MGI's, computing MGI's, and using MGI's. We will treat each one in turn.

Representing MGI's

The MGI is a property of a specialized function, in that it denotes the set of values on which the specialization can safely be applied. We can imagine a similar property for individual symbolic values, namely a value specification which, if substituted for the symbolic values's value specification, would cause the specializer (in the context of constructing the specialization) to perform the same reductions and return an equivalent code expression. We call this attribute the *domain specification*, and compute it incrementally during specialization.

In base FUSE, specializations are constructed on "new" symbolic values obtained by instantiating the argument specification, and these symbolic values never flow out of the body of the specialization; therefore, after the body of a specialization is complete, the domain specifications of its formals will not change. Thus, taking the domain specifications of the formals (elements of *Sval*) of a *code-fcn* object will yield its MGI.

The primary change made to the specializer is the addition of domain specifications to symbolic values. The domain specification of a symbolic value is drawn from the same type lattice as the value specification. Unlike the value specification and code expression attributes, which are immutable,[31] the domain specification of a particular symbolic value changes during the specialization process. Thus, the *identity* of symbolic values, which before was used only by the code generator, now becomes important in the specializer as well.

Domain Specifications and Domain Stores

To make the "stateful" nature of symbolic values explicit in our description, we introduce a new data structure, the *domain store*, which maps symbolic values to their domain specifications, and modify the specializer to single-thread the domain store through all of its operations.[32]

Domain specifications are drawn from the same type system, *Val*, as value specifications. When we embed them in symbolic values (in this case, indirectly using the domain store), it is useful to have structured (*e.g.*, pair- and closure-valued) domain specifications contain symbolic values rather than domain specifications. This is necessary so that changes to the domain specification of a symbolic value inside a structure are reflected in the domain specification of the entire structure. We accomplish this by having domain stores return elements of *Val′* instead of *Val*:

$$DomainStore : Sval \rightarrow Val'$$

To recover the domain specification of a symbolic value, we apply the projection function

[31] The value specification attribute is immutable in all versions of FUSE, while the internals of the code expression attribute are altered by the respecialization code in the "CFA FUSE" of Chapter 5 and Section A.5.

[32] As was the case with caches, the actual implementation doesn't single-thread a data structure representing the domain store, but merely side effects slots in the symbolic value objects.

domain-projection to the symbolic value and the current domain store. The projection function looks up the symbolic value in the domain store and, if the returned domain specification (element of Val') contains symbolic values, looks those up in the same domain store, etc., until all symbolic values have been dereferenced to elements of *Val*.

We define the "empty" domain store, which maps all symbolic values to $\top_{Val}$, in effect, saying that no reductions in the residual code expression of the symbolic values depended on the value specification of any of the input symbolic values; *i.e.*,

$$EmptyDomainStore = (\lambda\ s\ .\ \top_{Val})$$

Because domain specifications are computed incrementally, we need to define an update operation on domain stores. Because the domain specification of a symbolic value denotes the amount of information "used" about a particular symbolic value during specialization, a particular symbolic value's domain specification will only assume values lower in the type lattice over time. Thus, if a particular reduction makes use of the fact that a symbolic value s denotes a number, and the value of s has not been used before, the domain store should be updated to map s to $\top_{Number}$. However, if another reduction has already made use of the fact that s denotes the number 42, then the domain store (which maps s to 42) should be left unchanged. We can accomplish this by taking greatest lower bounds when updating the domain store

$update\text{-}dstore : DomainStore \rightarrow Sval \rightarrow Val' \rightarrow DomainStore$
$update\text{-}dstore\ dstore\ s\ d' =$
  $cond\ d' \sqsubseteq d \rightarrow dstore[d'/s],$
    $else \rightarrow dstore$
  $where\ d = (domain\text{-}projection\ s\ dstore)$

We use an explicit conditional rather than the least upper bound operator ($\sqcap$) because $\sqcap$ isn't defined on Val'—what code expressions would be used for symbolic values contained in the least upper bound? The comparison operator can easily be defined as an extension of the ordering on *Val*, just by taking the appropriate projection function from Val' to *Val*. (There is no problem with incompatible code expressions in the domain store since (1) any symbolic values in a domain specification are the same as those in the corresponding value specification, and thus have the same code fields, and (2) we never look at the code fields of domain specifications anyway).

$$
\begin{array}{lcl}
PE & : & Exp \rightarrow Env \rightarrow Cache \rightarrow Pgm \rightarrow Env \rightarrow DomainStore \rightarrow \\
 & & Sval \times Cache \times DomainStore \\
PE\text{-}seq & : & Exp^* \rightarrow Env \rightarrow Cache \rightarrow Pgm \rightarrow Env \rightarrow DomainStore \rightarrow \\
 & & Sval \times Cache \times DomainStore \\
PE\text{-}prim & : & Prim \rightarrow Sval^* \rightarrow DomainStore \rightarrow \\
 & & Sval \times DomainStore \\
PE\text{-}spec & : & Fcn \rightarrow Val^* \rightarrow Cache \rightarrow Pgm \rightarrow DomainStore \rightarrow \\
 & & Code\text{-}fcn \times Cache \times DomainStore
\end{array}
$$

Figure A.9: Signatures of major functions in the specializer with re-use

As we said earlier, we can compute the MGI of a complete specialization by taking the domain projections of the formals of the *code-fcn* object representing the specialization. If the specialization is incomplete, the domain specifications of the formals may yet be updated to values lower in the lattice. Therefore, we approximate the MGI of incomplete specializations (*e.g., code-fcn* with a dummy body) by the *value specifications* of the formals.

$$
\begin{array}{l}
specialization\text{-}MGI : Code\text{-}fcn \rightarrow DomainStore \rightarrow Val^* \\
specialization\text{-}MGI\ f\ dstore = \\
\quad cond\ (code\text{-}fcn\text{-}body\ f) = dummy \rightarrow (value\text{-}projection\ x_i)^n \\
\qquad\quad else \rightarrow (domain\text{-}projection\ x_i\ dstore)^n \\
\qquad where\ x_i^n = (code\text{-}fcn\text{-}formals\ f)
\end{array}
$$

The Revised Specializer

To compute domain specifications, we extend base FUSE by single-threading the domain store through some of the specializer's functions, as shown in Figure A.9. In addition, parts of the specializer that use information to perform reductions (*e.g.,* conditionals, primitives, `ABSCALL`s and cache lookup) are modified to update the domain store.

The code for constants and variables is basically unchanged, since returning a constant or looking up a variable cannot restrict the domain of specialization (remember that the code generator will inline the value of a ground-valued variable *only* when that value is used to perform a reduction elsewhere in the program (*i.e.,* when the domain specification is also a ground value); were this not the case, inlining a variable's value would affect the domain

store). Thus, we simply return the same domain store as was passed in. We show only the code for constants; variables are similar).

$$PE\ [\![\texttt{(CONST}\ k\texttt{)}]\!]\ env\ cache\ pgm\ senv\ dstore$$
$$= \langle (instantiate\ (constant\text{-}to\text{-}Val\ k)\ \texttt{no-code}),\ cache,\ dstore\rangle$$

The case of `let`-bindings is the same as before, except that the domain store is single-threaded in the obvious way:[33]

$$PE\ [\![\texttt{(LET}\ (x_i\ e_i)^n\ e\texttt{)}]\!]\ env\ cache\ pgm\ senv\ dstore$$
$$= PE\ e\ env'\ cache'\ pgm\ dstore'$$
$$where\ env' = env\ [s_i/x_i]^n$$
$$where\ \langle s_i^n, cache', dstore'\rangle = PE\text{-}seq\ e_i^n\ env\ cache\ pgm\ senv\ dstore$$

In the code for `if`, we see the first example of updating the domain specification of a particular symbolic value. When the predicate is known, we reduce the conditional at specialization time, and thus the residual program is only valid for forms where the predicate has the same truth value, forcing us to return a domain store where the predicate symbolic value's identity is mapped to a true or false value.[34] If the predicate cannot be reduced to a true or false value, we need not update the domain store to reflect the use of the predicate (the evaluation of the predicate and branch expressions may still update the domain store).[35]

$$PE\ [\![\texttt{(IF}\ e_1\ e_2\ e_3\texttt{)}]\!]\ env\ cache\ pgm\ senv\ dstore$$
$$= cond\ (istrue?\ (value\text{-}projection\ v_1)) \rightarrow PE\ e_2\ env\ cache'\ pgm\ senv\ dstore''$$
$$where\ dstore'' = (update\text{-}dstore\ dstore'\ s_1\ v_1),$$
$$(isfalse?\ (value\text{-}projection\ v_1)) \rightarrow PE\ e_3\ env\ cache'\ pgm\ senv\ dstore''$$
$$where\ dstore'' = (update\text{-}dstore\ dstore'\ s_1\ false),$$

[33] The helper function *PE-seq* performs similar single-threading, evaluating each form in a sequence on the domain store returned by evaluating the previous form. We won't show the code here.

[34] In our language, all values other than *false* are true. Since our type system cannot express negation, the domain specification for a nonfalse predicate is the most general type that dominates the predicate's type but not *false*. In our sample domain, *Val*, which has only scalars, pairs, and closures, the desired domain specification is simply the the value specification attribute (element of *Val'*) of the predicate symbolic value. Richer type domains would require more computation here.

[35] This version of `if` just uses $\top_{Val}$ as the return value for the unknown test case, in order to make the computation of domain stores as simple as possible. If we instead returned the generalization of s_2 and s_3 in this case, we would have to add dependency tracking (*c.f.* the `parents` attribute of Section 6.4.1). We will not treat that extension here.

$$
\begin{array}{l}
\quad else \rightarrow \langle (instantiate\ \top_{Val}\ \texttt{(IF}\ s_1\ \ s_2\ \ s_3\texttt{)}),\ cache''dstore''\rangle \\
\qquad where\ \langle\langle s_2, s_3\rangle, cache'', dstore''\rangle = \\
\qquad\quad PE\text{-}seq\ \langle e_2, e_3\rangle\ env\ cache'\ pgm\ senv\ dstore' \\
where\ \langle v_1, c_1\rangle = s_1 \\
\quad where\ \langle s_1, cache', dstore'\rangle = PE\ e_1\ env\ cache\ pgm\ senv\ dstore
\end{array}
$$

Primitives are similar to `let`, in that we must single-thread the domain store through the evaluation of the arguments. This is uninteresting, so we omit the code here. What *is* interesting is the computation of domain specifications when evaluating the primitives in the helper function *PE-prim*.

The idea is that primitives affect the domain of specialization only when they are reduced. So, in our sample primitives, `+` will update the domain specifications of its argument symbolic values when it is able to add them together, and will leave them alone otherwise. The primitive `cons` doesn't touch its arguments, so it always returns the domain store unaltered. The primitive `car`, if passed a pair, makes use of the fact that a particular pair was passed, but doesn't make use of the types of the components of that pair. This is denoted by setting the pair's domain specification to its value specification, so that if the pair's `car` or `cdr` field is used, this will be reflected in the pair's domain specification. If passed a non-pair, `car` makes no use of its input whatsoever.[36]

$$
\begin{array}{l}
PE\text{-}prim\ [\![\texttt{+}]\!]\ \langle s_1, s_2\rangle\ dstore \\
\quad = cond\ (number?\ v_1) \wedge (number?\ v_2) \rightarrow \langle\langle (+\ v_1\ v_2),\ \texttt{no-code}\rangle, dstore'\rangle \\
\qquad\qquad where\ dstore' = (update\text{-}dstore\ (update\text{-}dstore\ dstore\ s_1\ v_1)\ s_2\ v_2), \\
\qquad\quad else \rightarrow \langle\langle \top_{Number},\ \texttt{(PRIMCALL + }\ s_1\ \ s_2\texttt{)}\rangle, dstore\rangle \\
\qquad where\ \langle v_1, c_1\rangle = s_1,\ \langle v_2, c_2\rangle = s_2
\end{array}
$$

$$
\begin{array}{l}
PE\text{-}prim\ [\![\texttt{cons}]\!]\ \langle s_1, s_2\rangle\ dstore \\
\quad = \langle\langle (cons\ s_1\ s_2),\ \texttt{(PRIMCALL cons }\ s_1\ \ s_2\texttt{)}\rangle,\ dstore\rangle
\end{array}
$$

[36] Of course, in the real implementation, we might make use of the fact that `car`'s input was 42 to reduce the application of `car` to an error message, which would then return a domain store mapping the input symbolic value (denoting 42) to the highest type denoting 42 but no pairs (in this case, probably $\top_{Number}$). Similarly, operators like `integer?` or `+` can make use of only a portion of their arguments (`integer?` only needs to know if its input is an integer or not, while `+` can prove its result is an (unknown) integer if both of its inputs are (possibly unknown) integers.

$$
\begin{aligned}
&\mathit{PE\text{-}prim}\ [\![\texttt{car}]\!]\ \langle s_1\rangle\ \mathit{dstore}\\
&\quad = \mathit{cond}\ (\mathit{pair?}\ v_1) \rightarrow \langle (\mathit{car}\ v_1), \mathit{dstore}'\rangle\\
&\qquad\qquad \mathit{where}\ \mathit{dstore}' = (\mathit{update\text{-}dstore}\ \mathit{dstore}\ s_1\ v_1),\\
&\qquad\quad \mathit{else} \rightarrow \langle\langle \top_{\mathit{Val}},\ \texttt{(PRIMCALL car}\ s_1\texttt{)}\rangle,\ \mathit{dstore}\rangle\\
&\qquad \mathit{where}\ \langle v_1, c_1\rangle = s_1
\end{aligned}
$$

PE-prim is required to have the following property that if specializing a primitive p on argument a with value specification v and the empty domain store yields a domain store s mapping a to domain specification d, then specializing p on any argument a' with value specification v' where $v \sqsubseteq v' \sqsubseteq d$ and the empty domain store must return a domain store w' mapping a to d. More formally,

Property 16: *For all primitives p and argument vectors $a, a' \in \mathit{Sval}*$, whenever*

$$
\begin{gathered}
(\mathit{value\text{-}projection}\ a) \sqsubseteq (\mathit{value\text{-}projection}\ a') \sqsubseteq (\mathit{domain\text{-}projection}\ a\ w)\\
\mathit{where}\ \langle s, c, w\rangle = (\mathit{PE\text{-}prim}\ p\ a\ \mathit{EmptyDomainStore})
\end{gathered}
$$

it must be the case that

$$
\begin{gathered}
(\mathit{domain\text{-}projection}\ a\ w) = (\mathit{domain\text{-}projection}\ a'\ w')\\
\mathit{where}\ \langle s', c', w'\rangle = (\mathit{PE\text{-}prim}\ p\ a'\ \mathit{EmptyDomainStore})
\end{gathered}
$$

Informally, Property 16 says the following. Given some argument specification v, the specializer will reduce a primitive as completely as possible, which may produce a specialization which is applicable to some domain specification d larger than the argument specification. Given less information (but still enough to perform the reduction) v', the specializer will perform exactly the same amount of work, returning the same domain specification $d' = d$. If the specializer were to return a larger domain specification $d' \sqsupset d$, then it either (1) missed a possible reduction on v', or (2) could have returned the larger d' on v. Similarly, if the specializer were to return a smaller domain specification $d' \sqsubset d$, it either (1) missed a possible reduction on v, or (2) could safely return d instead of d'. (The case where d and d' are incomparable can't happen, because that would make $v \sqsubseteq v' \sqsubseteq d$ impossible).

User function calls are similar to primitives, in that both the evaluation of the arguments and the application of the function may update the domain store. The code for unfolding is largely unchanged, while the code for specialization has some extra work to do, because it must explicitly update the domain store. We do this whenever an operation makes use

of a value; the cases thus far were deciding conditionals and reducing primitives. When we decide to construct a call to a particular specialized procedure (*code-fcn*), we are using the argument values to decide which specialized procedure to invoke. That is, once we choose to invoke a particular specialized procedure, the new call expression is valid only for argument values which are contained in the MGI of the chosen specialization.[37]

We constrain the call site by updating the domain specifications of the argument symbolic values with the domain specifications of the formals (*i.e.,* the MGI of the specialization). The helper function *update-dstore-multiple* performs a sequence of updates to a domain store.[38]

$$
\begin{array}{l}
PE\,[\![\texttt{(USERCALL}\ f\ e_i^n\texttt{)}]\!]\ env\ cache\ pgm\ senv\ dstore \\
\quad = cond\ (unfold?\ f\ v_i^n) \rightarrow (PE\ b\ [s_i/x_i]\ cache'\ pgm\ senv\ dstore') \\
\qquad\qquad\qquad\qquad\qquad\qquad where\ x_i^n = (formals\ f\ pgm),\ b = (body\ f\ pgm), \\
\qquad\quad else \rightarrow \langle\langle \top_{Val},\ \texttt{(USERCALL}\ f'\ s_i^n\texttt{)}\rangle,\ cache'', dstore'''\rangle \\
\qquad\qquad\quad where\ dstore''' = (update\text{-}dstore\text{-}multiple \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad dstore'' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s_i^n \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (specialization\text{-}MGI\ f')) \\
\qquad\qquad\qquad\quad where\ \langle f', cache'', dstore''\rangle = PE\text{-}spec\ f\ w_i^n\ cache'\ pgm\ dstore' \\
\qquad\qquad\qquad\qquad where\ w_i^n = (get\text{-}general\text{-}args\ f\ v_i^n) \\
\qquad\quad where\ v_i = (value\text{-}projection\ s_i) \\
\qquad\qquad where\ \langle s_i^n, cache', dstore'\rangle = PE\text{-}seq\ e_i^n\ env\ cache\ pgm\ senv\ dstore
\end{array}
$$

The function *PE-spec* requires no revision beyond the usual single-threading of the domain store; the construction of the body of the specialization via unfolding will update the domain specifications of the formals appropriately, so that *specialization-MGI* will work properly.

---

[37] If the chosen specialization is incomplete, then we don't know its MGI yet—the domain specifications of the formal parameters might yet be updated to values lower in the lattice. In this Appendix, we take the rather conservative solution of using the *value specifications* of the formals as the MGI in this case. In the implementation, we relax this significantly if the residual `USERCALL` being constructed will be part of the body of the incomplete specialization in question (as opposed to part of the body of some other specialization)—in this case, it is permissible to use the overly general MGI since we know by construction that the `USERCALL` will only be invoked on parameters for which the specialization is applicable; we needn't restrict the applicability of the enclosing specialization at all.

[38] We omit the implementation of *update-dstore-multiple* due to some (largely uninteresting) technical issues relating to walking substructure in the argument symbolic values.

1. Single-thread the domain store (*PE*, *PE-seq*, *PE-prim*, *PE-spec*).
2. Update the domain store when a conditional is decided (*PE*).
3. Update the domain store when a residual `USERCALL` is constructed (*PE*).
4. Update the domain store when a `ABSCALL` is reduced (*PE*).

Figure A.10: Changes made to the specializer for computing MGIs

---

The `LAMBDA` special form doesn't examine any symbolic values whatsoever, so its code is also identical to that in base FUSE, except for the passing of the domain store. The `ABSCALL` form is similar to `LET`, primitive, and `USERCALL` forms in that the evaluation of the arguments may update the domain store. In this case, however, there's one extra twist. Since the head evaluates to a particular closure (remember: all `ABSCALL`s must be unfoldable), we are making use of this information[39] to perform the application, and must update the domain store to reflect this use.

$$
\begin{array}{l}
PE\,[\![\texttt{(ABSCALL}\ e_0\ e_i^n\texttt{)}]\!]\ env\ cache\ pgm\ senv\ dstore \\
\quad = (PE\ b\ env''\ cache'\ pgm\ dstore'') \\
\qquad\quad where\ dstore'' = (update\text{-}dstore\ dstore'\ s_0\ v_0) \\
\qquad\qquad where\ env'' = env'[s_i/x_i]^n \\
\qquad\qquad\quad where\ \langle x_i^n, b, env'\rangle = v_0, \\
\qquad\qquad\qquad where\ \langle v_0, c_0\rangle = s_0 \\
\qquad\qquad\qquad\quad where\ \langle\langle s_0, s_1, \ldots, s_n\rangle, cache', dstore'\rangle \\
\qquad\qquad\qquad\qquad = (PE\text{-}seq\ \langle e_0, e_1, \ldots, e_n\rangle\ env\ cache\ pgm\ senv\ dstore)
\end{array}
$$

Figure A.10 summarizes the changes we have just described.

---

[39] Of course, the application may or may not make use of particular values in the closure's environment. Updating the domain store to map the head symbolic value to the closure object (element of Val') says nothing about the use of values in the closure's environment. Instead, subsequent updating of the domain store on symbolic values bound in the environment will automatically be reflected whenever we examine the head's domain specification, since it *contains* the relevant symbolic values, which are dereferenced through the domain store each time *domain-projection* is applied. This is one example of why domain stores return elements of Val' rather than elements of *Val*.

The Revised Cache

The description above documents how the specializer *computes* domain specifications, but now how it it *makes use of* that information. As we saw in Chapter 6, domain specifications are useful for two purposes:

1. discarding redundant specializations, and
2. avoiding redundant specializations.

Both of these behaviors are implemented in the cache. The first, discarding redundant specializations after they are constructed, is implemented at cache *update* time. When *PE-spec* updates the cache (with a function name, argument specification, and *code-fcn*), it doesn't blindly add the new mapping to the cache. Instead, it examines all other (complete) specializations of the same function, and checks to see if any of them have the same MGI as the new specialization. If any prior specialization has the same MGI, *PE-spec* returns a cache that maps the *new* arguments to the *old* code object, and discards the new code object. Exactly how the search of the cache is performed is an (uninteresting) implementation detail, so we won't show the code for cache update here. All that is important is that specialization bodies with MGI equal to the MGI of an existing specialization of the same function may be discarded.

The second behavior, avoiding redundant specializations, occurs at cache *lookup* time. Instead of just comparing values specifications (as is done in base FUSE), the lookup routine scans the cache, looking for a (complete) specialization with argument and domain specifications such that it can prove that the new specialization would have the same MGI. This can be accomplished in several ways (*c.f.* Section 6.4.3); we will consider only the basic one of Section 6.3.3. The basic criterion says that an existing specialization can be (optimally) reused whenever

$$v \sqsubseteq v' \sqsubseteq d$$

where v and d are the argument specification (tuple of value specifications) and MGI (tuple of domain specifications) of the existing specialized procedure, and v' is the argument specification of the new call site. The code for checking this inequality for all relevant specializations in the cache is uninteresting, and will not be shown here. What is important is that an existing specialization can be re-used at a call site even if its argument specification

differs from the argument specification at the call site, as long as the call site's argument specification lies between the specialization's MGI and the specialization's argument specification.

A.6.2 Correctness

We will prove two properties. First, the MGIs computed by our specializer really are safe approximations to the domain of specialization; that is, a specialized procedure will return the same value as the source procedure for all values specified by the MGI. This allows us to discard specializations with identical MGI's. Second, our preemptive re-use criterion is optimal; that is, when the $v \sqsubseteq v' \sqsubseteq d$ criterion (described above and in Section 6.3.3 holds, constructing a new specialization on v' is guaranteed to produce redundant code.

Before we begin, however, we must make one alteration to our existing formalism. In base FUSE, the code generator is permitted to use the value (rather than the code expression) of any symbolic value with a ground value specification. When the re-use mechanism is added, this is no longer the case; if the ground value was never used during specialization, then it might be the case that the code expression will evaluate to *different* ground values at runtime. Instead, we restrict the code generator to inline ground-valued *domain specifications* instead of ground-valued *value specifications*, and we alter the meaning of the graph evaluator $\mathcal{E}'$ to reflect this change. However, to find the domain specification of a symbolic value, the evaluator/code generator needs to have access to the domain store, so we add a new parameter to $\mathcal{E}'$, yielding the signature

$$\mathcal{E}' = Sval \rightarrow SchemeEnv \rightarrow DomainStore \rightarrow SchemeValue.$$

Safety

What we want to show is that, at each step in the specialization process, the domain store safely approximates the set of values for which the residual code expression will compute the same value as the source expression.

Property 17: *Given any expression $e \in Exp$ which is part of a program $p \in Pgm$, specialization-time environment $r \in Env$ defined on all free variables in e, cache $c \in Cache$, specialization-time environment $h \in Env$ defined on all free variables in (code-projection r), and domain store $d \in DomainStore$ then for any Scheme environments $r' \in (C\ (\textit{domain-projection}\ r\ d'))$, $h' \in (C\ (\textit{domain-projection}\ h\ d'))$, and $h'' = (\textit{spread-env}\ h\ h')$,*

$$(\mathcal{E}'\, s\, h''\, d') = (\mathcal{E}\ e\, r'\, p)$$
$$\textit{where } \langle s, c', d' \rangle = (PE\, e\, r\, c\, p\, h\, d).$$

provided that

1. *for all free variables x in e,*

$$(\mathcal{E}'\,(r\,x)\,h''\,d) = (r'\,x)$$

2. *for all function names f in p and argument specifications a of appropriate arity, $(c\ f\ a)$ is either undefined or returns a specialization g such that*

$$(\mathcal{E}'\,(\textit{code-fcn-body}\ (\textit{completion}\ g))\,(\textit{spread-env}\,t\,t')) = (\mathcal{E}\ (\textit{body}\ f)\,t'\,p)$$

where $t = [(\textit{code-fcn-formals}\ g)_i/(\textit{formals}\ f)_i]^n)$,
$t' \in (C\ (\textit{value-projection}\ [(\textit{specialization-MGI}\ g\ d)_i/(\textit{formals}\ f)_i]^n))$.

Proof: Assume, for the moment, that *PE-Spec* uses the "base" cache; that is, not discarding or preemptively re-using specializations (we will add discarding at the end of this proof, and preemptive re-use in Property 18). Since no reduce/residualize decisions have changed, the generated code is identical to that under base FUSE,[40] and the proof of Property 6 still holds (*i.e.*, the generated code is general enough to run on all instances of the initial environment). We will show that, if that is the case, the generated code is also general enough to run in all instances of the initial environment where value specifications have been replaced by domain specifications.

[40] Modulo the fact that irreducible `IF`s now return a value specification of $\top_{Val}$ instead of the generalization of the arms. This changes the proof of Property 5 only trivially, so we won't worry about it here.

We prove this property by induction on the number of calls to PE during the specialization of e.

Hypothesis: Property 17 is true for e', r', c', p, h', and d' where executing $(PE\ e'\ r'\ c'\ p\ h'\ d')$ takes fewer steps than executing $(PE\ e\ r\ c\ p\ h\ d)$.

Base cases: If e is a constant, the property is trivially true (evaluating a constant to a constant doesn't affect the applicability of the residual constant). If e is a variable, dereferencing it doesn't depend on the value specification of any symbolic value (provided, of course, that the code generator (and graph evaluator) doesn't inline ground value specifications of symbolic values unless the domain specification is also ground). Thus, it is correct to just return the initial domain store.

Inductive cases:

- `LET`: The property is true by hypothesis for all initializers and the body. Because the domain store is single-threaded through these evaluations, it reflects all restrictions due to reductions performed in the initializers and body (*i.e.,* it has been updated so that the domain specifications of all symbolic values referenced in the initializers and body denote a set of values for which the residual expression computes the same function as the source expression did). Since `let` forms are always reducible, no other restrictions are necessary.

- `IF`: The property is true by hypothesis for the predicate and the arm(s) that get specialized. Because the domain store is single-threaded through these evaluations, it reflects all restrictions due to reductions performed in the predicate and arm(s). If the consequent or alternative was chosen, the residual expression is valid only for equivalent predicates; the domain store is updated to reflect this.

- `PRIMCALL`: The property is true by hypothesis for the arguments; because of single-threading, the domain store reflects all restrictions due to reductions performed in the arguments. In addition, if the primitive is reduced, the domain store is updated to reflect the fact that the residual expression is valid only for equivalent arguments.

- `USERCALL`: The property is true by hypothesis for the arguments; because of single-threading, the domain store reflects all restrictions due to reductions performed in the arguments. If the call is unfolded, the property is also true (by hypothesis) for the unfolded body. If the call is specialized, there are two cases. If the cache defines

a specialization f, then (*specialization-MGI* f) correctly defines the conditions under which f is applicable (by explicit assumption in the property statement), and the domain store returned from the `USERCALL` reflects these restrictions. If the cache doesn't define the specialization, then the MGI computed during construction of the specialization is correct (by hypothesis), and once again, the returned domain store will reflects its restrictions.

- `LAMBDA`: Since the closure is not specialized, this constructs no residual code, and the property holds trivially.
- `ABSCALL`: The property is true by hypothesis for the head and arguments; because of single-threading, the domain store reflects all restrictions due to reductions performed in the head and arguments. Because the call is unfolded, we are making use of the fact that the head evaluated to a closure with a particular body (allowing us to unfold the body), and we may make use of the value specifications of symbolic values in the closure's environment and in the arguments. The former is noted by updating the domain store for the head symbolic value with the closure object; the latter is accomplished by single-threading the resultant domain store through the unfolding of the body. By the hypothesis, the resulting domain store will reflect all restrictions due to reductions in the head, arguments, application, and body.

■

Now that we know domain specifications safely approximate the set of values for which a residual expression computes the same function as a source expression, we can use them to discard redundant specializations. In particular, if two specializations of the same function have identical domain specifications (MGIs), then these specializations are applicable to the same sets of values, and thus can be freely substituted for one another. That means it is permissible to discard a newly-constructed specialization whenever its MGI is identical to that of an existing specialization of the same function. This validates the first change made to the caching mechanism on Page 331.

Optimality

The other change made to the cache was in the lookup routine. We know from Property 17 that it is safe to build a residual call to a specialization g whenever the argument specification a of the call site is contained in the MGI of the specialization (*i.e.,* whenever

$a \sqsubseteq (\mathit{specialization\text{-}MGI}\, g)$). However, this is only a safety criterion; it doesn't guarantee that g is the *best* specialization we could produce for a (*e.g.,* constructing a new specialization g' on a might give $g' \sqsubseteq (\mathit{specialization\text{-}MGI}\, g)$, meaning g' is more optimized than g). Without some other optimality criterion, the only way to be sure that g is the best specialization is to construct a new specialization g' on a, then compare the MGIs of the two specializations. If the MGIs are the same, then using g is optimal, and g' can be discarded. Unfortunately, this solution requires computing a new specialization g' for each argument specification a.

In Chapter 6, we developed several alternate optimality criteria which allowed us to avoid building new specializations in some cases. The simplest one, which we will discuss here, reuses g on a whenever[41]

$$(\mathit{value\text{-}projection}\, (\mathit{code\text{-}fcn\text{-}formals}\, g)) \sqsubseteq a \sqsubseteq (\mathit{specialization\text{-}MGI}\, g)$$

This method relies on the following property:

Property 18: *Consider specializing any expression $e \in Exp$ which is part of a program $p \in Pgm$ on any specialization-time environment $r_1 \in Env$ defined on all free variables in e, cache $c \in Cache$, "enclosing specialization environment" $h \in Env$, and domain store d_1. Given the domain store d_1' computed by $(PE\ e\ r_1\ c\ p\ h\ d_1)$, the domain store d_2' returned by $(PE\ e\ r_2\ c\ p\ h\ d_2)$ where*

$$(\mathit{value\text{-}projection}\ r_1) \sqsubseteq (\mathit{value\text{-}projection}\ r_2) \sqsubseteq (\mathit{domain\text{-}projection}\ r_1\ d_1')$$

and

$$(\mathit{domain\text{-}projection}\ r_1\ d_1) \sqsubseteq (\mathit{domain\text{-}projection}\ r_2\ d_2)$$

has the property

$$(\mathit{domain\text{-}projection}\ r_1\ d_1') \sqsubseteq (\mathit{domain\text{-}projection}\ r_2\ d_2')$$

[41] It only makes sense to apply this criterion to complete specializations in the cache, since for incomplete specializations, the argument specification and MGI are considered to be the same (*c.f.* definition of *specialization-MGI*).

This property is just common sense. Given some value specification a, the specializer is only able to perform some number of reductions, yielding a specialized expression valid for some more general domain specification b. Then we give the specializer another specification a' where $a \sqsubseteq a' \sqsubseteq b$, Since a' denotes less information than a, we can't perform any more reductions than we did given a; at the same time, we have at least as much information as was actually used (denoted by b), so we will be able to perform all of the same reductions we did before. Thus, we expect $b = b'$. That would be the case, except that our termination mechanisms aren't perfect, and sometimes forego reductions by abstracting argument specifications unnecessarily; we will only be able to show $b \sqsubseteq b'$, but that's all we need.

Proof: By induction on the number of calls to *PE* during the specialization of e.

Hypothesis: Property 18 is true for e', r'_1, r'_2, c', p, h'_1, h_2', d'_1, and d'_2 where executing $(PE\ e'\ r'_1\ c'\ p\ h'_1\ d'_1)$ takes fewer steps than executing $(PE\ e\ r_1\ c\ p\ h_1\ d_1)$, and executing $(PE\ e'\ r'_2\ c'\ p\ h'_2\ d'_2)$ takes fewer steps than executing $(PE\ e'\ r_2\ c'\ p\ h_2\ d_2)$.

Base cases: Trivially true for both constant and variables, because they don't alter the domain store.

Inductive cases: The property is true by hypothesis for all evaluations of argument subforms (*i.e.,* initializers in `LET`, predicate in `IF`, arguments in `PRIMCALL` and `USERCALL`, head and arguments in `ABSCALL`).

We treat the remaining cases below. First, however, we establish a useful relation which is true for all evaluation of the argument subforms listed above. The symbolic value returned by specializing each argument is either (1) constructed during the specialization of the argument expression, or (2) already present in the environment, and merely selected by the argument expression. Any argument symbolic value constructed during specialization will have the same value specification under r_1 and r_2 (because the domain specifications of the inputs force those inputs to be the same under r_1 and r_2), while all argument symbolic values taken from the environment will have domain specifications in d_1 representing the largest set of values for which the residual code returned by the entire expression e (containing the argument expression) will be valid. Thus, the argument symbolic value obtained from r_2 will lie between the domain and value specifications of the corresponding symbolic value in r_1.

Thus, for each argument form a, we have $v_1 \sqsubseteq v_2 \sqsubseteq w_1$, where v_1 and v_2 are the value specifications computed by specializing a under r_1 and r_2, respectively, and w_1 is the domain

specification of the symbolic value obtained by specializing a under r_1, examined after the specialization of the entire outer form e under r_1 is complete.

Since the single-threaded argument evaluation performed by *PE-spec* preserves the property $(\textit{domain-projection}\, r_1\, d_1) \sqsubseteq (\textit{domain-projection}\, r_2\, d_2)$, allowing the hypothesis to be applied repeatedly to each evaluation in the series, the property above on argument values generalizes to entire argument tuples under the usual extension of a partial ordering on values to a partial ordering on tuples.

- `LET`: Since $a_1 \sqsubseteq a_2 \sqsubseteq b_1$, and the domain store returned by the `LET` is the same as that returned by the body, the environments r_1' and r_2' used to specialize the body when the `LET` is specialized under r_1 and r_2, respectively, will obey the constraint $(\textit{value-projection}\, r_1') \sqsubseteq (\textit{value-projection}\, r_2') \sqsubseteq (\textit{domain-projection}\, r_1, d_1)$. Thus, the hypothesis applies to the specialization of the body.

- `IF`: Since $a_1 \sqsubseteq a_2 \sqsubseteq b_1$, where a_1 is the value specification of the predicate under r_1, a_2 is the value specification under r_2, and b_1 is the domain specification of predicate after the evaluation of the entire `IF` expression, we have the following:

  If a_1 has a known truth/falsity, then b_1 will reflect the use of this in reducing the `IF` to one of its branches, and thus a_2 will have the same known truth/falsity. The `IF` will be reduced to the same branch under r_2; evaluation of the chosen branch takes place in the same environment as that used to evaluate the `IF`, and an equivalent domain store in both cases, so the hypothesis applies.

  If a_1 has an unknown truth/falsity, the `IF` will not be reduced, so b_1 will also have an unknown truth/falsity. Neither branch will update the domain specification of the predicate to any element of *Val* whose truth/falsity is known. By the relation above, a_2 will also have an unknown truth/falsity, meaning that the `IF` will not be reduced under r_2; the hypothesis applies to both branches.

- `PRIMCALL`: Since $a_1 \sqsubseteq a_2 \sqsubseteq b_1$, where a_1 is the value specification of the arguments under r_1, a_2 is the value specification of the arguments under r_2, and b_1 is the domain specification of the arguments under r_1, examined after the evaluation of the entire `PRIMCALL` expression, we can guarantee that $d_1 = d_2$ by applying Property 16 and the properties of greatest lower bounds.

- `USERCALL`: The relation $a_1 \sqsubseteq a_2 \sqsubseteq b_1$, where the variables denote specifications of all argument expressions, holds. We consider the following cases:

  - Function is unfolded on both a_1 and a_2: In this case, the hypothesis applies to the unfolding of the body.
  - Function is specialized on both a_1 and a_2: By monotonicity of the function *get-general-args* (Property 4), and the fact that all uses of formal parameters are reflected in the domain specifications of the actuals, we find that the hypothesis is applicable to the computation of the specialized function bodies.
  - Function is unfolded on a_1 and specialized on a_2: Under an ideal termination strategy, this never happens. We unfold a function because we know enough about its induction variable(s) to determine the number of iterations. The use of that information is reflected in the domain specifications of the induction variables (or the values contributing to their construction), and thus the induction variables are constrained to have the same values in a_1 and a_2, in which case the termination mechanism *should* choose to unfold again. FUSE's termination mechanisms generally do, but none of the properties we have given here guarantee this; as long as a_2 is more general than a_1, the oracle *unfold?* is allowed to change its mind.

    If this does occur, the specialization is constructed on arguments more general than a_2, so it may be that not all of the induction variable values in a_2 are used, in which case the returned domain store under a_2 may be more general than that under a_1. But it certainly won't be more specific, which is all we need to rule out.
  - Function is specialized on a_1 and unfolded on a_2: This can't happen, as it would violate the monotonicity of the function *unfold?* (Property 3).

- `LAMBDA`: The closure is never specialized; the property is trivially true since the domain store is not updated.

- `ABSCALL`: The call is always unfolded (*i.e.,* the head is known under r_1). By the relation $a_1 \sqsubseteq a_2 \sqsubseteq b_1$, where the variables denote specifications of all subexpressions including the head, we find that the head must also be known under r_2, or unknown in both cases. The body will be same in both unfoldings, and the environments

and domain stores used in performing the unfolding are such that the hypothesis is applicable.

■

We have shown that a specialization g of a function f constructed on value specification a will have an MGI at least as large as that of another specialization g' of f' constructed on a' whenever $a' \sqsubseteq a \sqsubseteq (\mathit{specialization\text{-}MGI}\, g')$. Since the new specialization is applicable to at least as many values as the existing specialization, it can't be any more optimized, so it's proper to preemptively reuse the existing one.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, 1991.

[4] L. Andersen. Self-applicable C program specialization. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 54–61, June 1992.

[5] L. Andersen and C. Gomard. Speedup analysis in partial evaluation (preliminary results). In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 1–7, 1992.

[6] W.-Y. Au, D. Weise, and S. Seligman. Generating compiled simulations using partial evaluation. In *Proceedings of the 28th Design Automation Conference*, pages 205–210. IEEE, June 1991.

[7] L. Beckman et al. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7(4):291–357, 1976.

[8] A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1989. Published as Artificial Intelligence Laboratory Technical Report TR-1144.

[9] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990.

[10] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[11] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, pages 70–87. Springer-Verlag, LNCS 432, 1990.

[12] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[13] A. 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.

[14] A. Bondorf and O. Danvy. Automatic autoprojection for recursive equations with global variables and abstract data types. DIKU Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990.

[15] A. Bondorf, N. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[16] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[17] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[18] C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992. Published as technical report STAN-CS-92-1420.

[19] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *LISP and Symbolic Computation*, 4(3):283–310, July 1991.

[20] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990.

[21] C. Consel. New insights into partial evaluation: the SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming*, pages 236–246, Nancy, France, 1988. Springer-Verlag, LNCS 300.

[22] C. Consel. *Analyse de programmes, Evaluation partielle et Génération de compilateurs*. PhD thesis, Université de Paris 6, Paris, France, June 1989. 109 pages. (In French).

[23] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, 1990.

[24] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[25] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, pages 88–105. Springer-Verlag, LNCS 432, 1990.

[26] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, (LNCS 523)*, pages 496–519, Cambridge, MA, August 1991. ACM, Springer-Verlag.

[27] C. Consel and O. Danvy. Static and dynamic semantics processing. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 14–24. ACM, January 1991.

[28] C. Consel and O. Danvy. Partial evaluation: Principles and perspectives. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, 1993. (tutorial presented at the conference, to appear).

[29] C. Consel and S. Khoo. Parameterized partial evaluation. In *SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, Toronto, Canada. (SIGPLAN Notices, vol. 26, no. 6, June 1991)*, pages 92–105. ACM, 1991.

[30] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1992. (draft, to appear).

[31] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *IEEE International Conference on Computer Languages*, Oakland, CA, April 1992. IEEE.

[32] O. Danvy. Personal communication. March 1991.

[33] A. De Niel. Partial evaluation: Its application to compiler generator* generation. Technical Report CMU-CS-88-166, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[34] A. De Niel, E. Bevers, and K. De Vlaminck. Program bifurcation for a polymorphically typed functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 142–153. ACM, 1991.

[35] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[36] J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *ACM Conference on Lisp and Functional Programming*, pages 331–347. ACM, 1984.

[37] A. Ershov. On the essence of compilation. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[38] A. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*. OHMSHA Ltd. and Springer-Verlag, 1988.

[39] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

[40] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[41] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[42] C. Ghezzi, D. Mandrioli, and A. Tecchio. Program simplification via symbolic interpretation. In S. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science. Fifth Conference, New Delhi, India. (Lecture Notes in Computer Science, Vol. 206)*, pages 116–128. Springer-Verlag, 1985.

[43] R. Glück. Towards multiple self-application. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 309–320. ACM, 1991.

[44] C. Gomard and N. Jones. Compiler generation by partial evaluation: A case study. *Structured Programming*, 12:123–144, 1991.

[45] C. Gomard and N. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[46] M. A. Guzowski. Towards developing a reflexive partial evaluator for an interesting subset of LISP. Master's thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, January 1988.

[47] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 4(4):501–538, October 1985.

[48] T. Hansen. Properties of unfolding-based meta-level systems. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 243–254. ACM Press, 1991.

[49] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, 1977. Published as Linköping Studies in Science and Technology Dissertation No. 14.

[50] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation: An International Journal 2:3/4:*, pages 179–396, 1989.

[51] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[52] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Springer-Verlag, 1991.

[53] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming (LISP Pointers vol. 5, no. 1, January-March 1992)*, pages 205–215, June 1992.

[54] C. K. Holst. Finiteness analysis. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*, pages 473–495. ACM, Springer-Verlag, 1991.

[55] C. K. Holst and J. Hughes. Towards binding time improvement for free. In S. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 83–100. Springer-Verlag, 1991.

[56] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[57] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91 Conference Proceedings*, pages 21–38. Springer-Verlag LNCS, July 1991.

[58] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 351–363, August 1986.

[59] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 154–165. ACM, 1991.

[60] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jouannaud, editor, *Conference on Functional Programming and Computer Architecture, Nancy*. Springer-Verlag (LNCS 201), 1985.

[61] N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 296–306. ACM, 1986.

[62] N. 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. North-Holland, 1988.

[63] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. 1993. (in progress).

[64] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, pages 124–140. Springer-Verlag, LNCS 202, 1985.

[65] 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.

[66] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, 1986.

[67] K. M. Kahn. A partial evaluator of Lisp programs written in Prolog. In M. V. Caneghem, editor, *First International Logic Programming Conference*, pages 19–25, Marseille, France, 1982.

[68] A. Kanamori and D. Weise. An empirical study of an abstract interpretation of Scheme programs. Unpublished manuscript, 1991.

[69] M. Katz. Personal communication. September 1991.

[70] M. Katz and D. Weise. Towards a new perspective on partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program*

Manipulation, pages 29–37, San Francisco, CA, 1992. Proceedings available as YALEU/DCS/RR-909.

[71] S. Khoo and R. Sundaresh. Compiling inheritance using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 211–222. ACM, 1991.

[72] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[73] S. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[74] A. Lakhotia and L. Sterling. ProMiX: A Prolog partial evaluation system. In L. Sterling, editor, *The Practice of Prolog*, chapter 5, pages 137–179. MIT Press, 1991.

[75] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An architecture for an open compiler. In *IMSA Workshop on Meta-Level Architectures and Reflection*, November 1992.

[76] J. Launchbury. *Projection factorisations in partial evaluation.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[77] J. Launchbury. Self-applicable partial evaluation without s-expressions. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*, pages 145–164. ACM, Springer-Verlag, 1991.

[78] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In Berkeley and Bobrow, editors, *The Programming Language Lisp*, pages 204–219. MIT Press, Cambridge, MA, 1964.

[79] K. Malmkjær. Predicting properties of residual programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 8–13, 1992. Proceedings available as YALEU/DCS/RR-909.

[80] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992.

[81] T. Mitchell, M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[82] T. Mogensen. Partially static structures. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[83] T. Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhangen, Copenhagen, Denmark, March 1989.

[84] B. R. Murphy. A type inference system for FL. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[85] M. Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 122–128, 1989.

[86] J. Peterson. Untagged data in tagged languages: choosing optimal representations at compile time. In *Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 89–99, 1989.

[87] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall International, 1987.

[88] J. Rees, W. Clinger, et al. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4(3):1–55, 1991.

[89] L. H. Rodriguez Jr. Coarse-grained parallelism using metaobject protocols. Master's thesis, MIT, 1991. Published as Xerox PARC Technical Report SSL-91-06.

[90] S. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

[91] E. Ruf. Errata for "Using types to avoid redundant specialization". *ACM SIGPLAN Notices*, 27(1):37–39, 1992.

[92] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 321–333. ACM, 1991.

[93] E. Ruf and D. Weise. Avoiding redundant specialization during partial evaluation. Technical Report CSL-TR-92-518, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[94] E. Ruf and D. Weise. Improving the accuracy of higher-order specialization using control flow analysis. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 67–74, San Francisco, CA, 1992. Proceedings available as YALEU/DCS/RR-909.

[95] E. Ruf and D. Weise. On the specialization of online program specializers. Technical report, Computer Systems Laboratory, Stanford University, Stanford, CA, July 1992.

[96] E. Ruf and D. Weise. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[97] E. Ruf and D. Weise. Preserving information during online partial evaluation. Technical Report CSL-TR-92-517, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[98] B. Rytz and M. Gengler. A polyvariant binding time analysis. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 21–28, 1992.

[99] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 377–398. MIT Press, 1990.

[100] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, March 1991. Report TRITA-TCS-9101, 170 pages.

[101] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, MIT, Cambridge, MA, August 1984. Published as MIT/LCS/TR-324.

[102] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[103] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

[104] P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, 1988. Published as DIKU Student Report 88-7-2.

[105] O. Shivers. *Control Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. Published as technical report CMU-CS-91-145.

[106] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[107] D. Smith and T. Hickey. Partial evaluation of a CLP language. In *Logic Programming: Proceedings of the 1990 North American Conference*, pages 119–138. MIT Press, 1990.

[108] G. L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1978.

[109] B. Steensgaard. Personal communication. October 1991.

[110] R. Sundaresh. Building incremental programs using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991)*, pages 83–93. ACM, 1991.

[111] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[112] V. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[113] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36(3):401–412, October 1988.

[114] D. Weise. Graphs as an intermediate representation for partial evaluation. Technical Report CSL-TR-90-421, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990.

[115] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 165–191, Cambridge, MA, August 1991. ACM, Springer-Verlag.

[116] D. Weise and E. 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.

[117] D. Weise and S. Seligman. Accelerating object-oriented simulation via automatic program specialization. Technical Report CSL-TR-92-519, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[118] J. Young and P. O'Keefe. Experience with a type evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581. North-Holland, 1988.