Preserving Information during Online Partial Evaluation*

Erik Ruf and Daniel Weise

Technical Report: CSL-TR-92-517
(also FUSE Memo 92-8)

April, 1992

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

Abstract

The degree to which a partial evaluator can specialize a source program depends on how accurately the partial evaluator can represent and maintain information about runtime values. Partial evaluators always lose some accuracy due to their use of finite type systems; however, existing partial evaluation techniques lose information about runtime values even when their type systems are capable of representing such information. This paper describes two sources of such loss in existing specializers, solutions for both cases, and the implementation of these solutions in our partial evaluation system, FUSE.

Key Words and Phrases: Partial Evaluation, Program Specialization, Online Specialization, Abstract Interpretation, Control Flow Analysis.

*This research has been supported in part by NSF Contract No. MIP-8902764, and in part by Advanced Research Projects Agency, Department of Defense, Contract No. N0039-91-K-0138. Erik Ruf is funded by an AT&T Foundation Ph.D. Scholarship.

Introduction

A *program specializer* (also called a *partial evaluator*) transforms a program and a specification restricting the possible values of its inputs into a *specialized* program that operates only on those input values satisfying the specification. The specializer uses the information in the specification to perform some of the program's computations at specialization time, producing a specialized program that performs fewer computations at runtime, and thus runs faster than the original program.

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. Performable computations are not always performed: to guarantee the termination of the specializer, and to increase sharing in residual programs, the specializer performs *folding* [9] operations by recursively specializing parts of the program, and building residual invocations of these *specializations*. These specializations may be invoked from multiple residual call sites, allowing them to be re-used. Program specializers also perform *generalization* [43, 45, 37, 6] operations, ignoring some specialization-time information and prohibiting some reductions in order to guarantee that only a finite number of specializations are built.

Ideally, a program specializer would perform as many reductions as possible[1] (while still terminating) at specialization time, so that they wouldn't have to be performed at run time. 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 paper 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 primitives, as we consider this part of the type system as well. Instead, this paper identifies and treats 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.

As a simple example of where existing methods fall short, consider code that maintains a store represented as an association list (Figure 1). The procedures `lookup-in-store` and `update-store` access and update (functionally) a store, while the procedure `process-updates` accepts a list of

[1] 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 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.

```
(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 1: Code to access and update (functionally) a store

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 . <any value>) (b . <any value>))`[2] and an unknown `name` and `value` will return a store of the same form. This information (the shape 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 . <any value>) (b . <any value>))` 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 methods described in this paper overcome this problem, and compute the desired return approximation for `process-updates`.

A related problem arises when higher-order procedures are introduced. The specializer must ensure that any specialization of such a procedure is sufficiently general to be applicable at all call sites it might reach at runtime. Thus, when constructing the body of the specialization, (by

[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 `<any value>`, `<any 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 `(<any value> . <any integer>)`.

unfolding the procedure's body) the specializer must bind the formal parameters to approximations that safely approximate all values that might be passed at runtime. Existing higher-order specializers [4, 12, 17, 45][3], do not compute accurate approximations for parameters of specializations of higher-order procedures, but instead approximate each parameter by "any value." This will certainly result in a sufficiently general specialization (since "any value" approximates all runtime values), but may forego many reductions which could have been performed had a better argument approximation been available. This paper describes a technique for computing such approximations, and shows its use on several examples.

This paper has five sections. The first describes program specialization, its *online* variant, and our partial evaluation system, FUSE. Section 2 introduces two examples of programs which specialize poorly under existing schemes, but which are handled well by our new mechanisms. Section 3 describes and solves the problem of computing approximations to values returned by specializations of first-order procedures, while Section 4 treats the problem of computing approximations to values passed as parameters to specializations of higher-order procedures. The final section describes related work in program specialization, type inference, and control flow analysis.

1 Background

In this section, we describe program specialization, and one if its implementations, *online program specialization*, using our system, FUSE, as a concrete example.

1.1 Defining 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:

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

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

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 $\perp_V$); a stricter definition would preserve the termination properties of the original function definition. Second, the residual function definition takes the same formal parameters as

[3] [38] and [18] treat higher-order languages, but cannot build specializations of higher-order procedures, so the problem does not arise.

the original function; we consider *reparameterization* [37] behavior such as the removal of completely known parameters or *arity raising* [34] 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. In order to describe and compare various strategies for performing program specialization, we must take a more operational view of specialization.

1.2 Online Program Specialization

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

Online program specializers make at least some of these reduce/residualize decisions at specialization time, while *offline* specializers make all such decisions prior to specialization time (for further comparisons of online and offline techniques, see [36, 7, 25]. The problems we will consider in this paper arise in both online and offline specialization, but our solutions are specific to online methods.

The structure of an online specializer is similar to that of an interpreter, except that online specializers represent unknown values explicitly, and construct a residual program instead of (or in addition to) returning a result value. Most of the reduce/residualize decisions are simple: expressions are reduced when enough information is available, and residualized otherwise; *e.g.,* residual code is generated for `if` expressions with unknown tests, `call` expressions with unknown heads, and primitive expressions with unknown parameters in strict positions. *Folding* [9] operations are performed by leaving some potentially reducible `call` expressions residual, building a residual call to a specialized version, or *specialization*, of the function being called. Such specializations can be invoked from multiple sites in the residual program, allowing the construction of loops and the sharing of code. Strategies for building and caching specializations vary; virtually all online specializers can build multiple specializations of a single function, a behavior known as *polyvariant specialization* [8].

To obtain an online specializer from an interpreter, three extensions are necessary. First, the value domain of the interpreter must be extended to represent values which are unknown at specialization time. Many different extensions are possible; they range from type lattices with a single "unknown" type added at the top of the interpreter's type lattice, through systems which can represent partial information about structures and scalar arithmetic types, to systems which can represent sets of concrete values and have recursive type descriptions. Second, the interpreter and its primitives must be extended to handle this richer value domain. As each expression is processed, the specializer must not only build residual code if necessary, but must also compute an approximation to the expression's return value, so that expressions consuming that value can be processed. Finally, the handling of `call` expressions must be extended to handle the construction and re-use of specializations (this includes choosing between unfolding and residualizing calls, determining which actual parameter values to use when building specializations, and when to re-use existing specializations).

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," 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," or by "any value," the `if` will not be seen to be reducible, and will not be reduced.

Sometimes, information loss is unavoidable. 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 approximations 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 "an integer" or "any value," 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 [15], and the `if`-handler code on p. 11 of [46]). 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 [45], Section 2.2 of [36], and [43].) 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.

In this paper, we will show two instances where present systems lose information unnecessarily, along with means for recovering this information.

1.3 FUSE: an online program specializer

This section briefly describes our program specializer, FUSE; for further documentation, see [45, 44].

FUSE is a polyvariant online program specializer for a side-effect-free subset of the programming language Scheme [33], with atoms, pairs, and higher-order procedures, but no vectors, input/output, first-class continuations, or `apply`. FUSE also enforces some minor syntactic constraints which are not of importance here: the syntax and semantics of the language treated in this paper are essentially those of Scheme.

1.3.1 Type System

Like other specializers, FUSE represents possibly infinite sets of runtime values using approximations drawn from a type system. FUSE's type system is obtained by extending the value domains of the Scheme subset it processes, shown in Figure 2. To describe sets of values, FUSE

Num			numbers
$Bool$	$=$	$true + false$	booleans
Sym			symbols
Nil	$=$	nil	empty list
$Pair$	$=$	$Val \times Val$	pairs
$Func$	$=$	$Val \rightarrow Val$	function values
Val	$=$	$Num + Bool + Sym + Nil + Pair + Func$	Values

Figure 2: Value domains for a subset of Scheme

`NumType`	`::=`	`top-num` | $n \in Num$
`BoolType`	`::=`	`top-bool` | $b \in Bool$
`SymType`	`::=`	`top-sym` | $s \in Sym$
`NilType`	`::=`	nil
`PairType`	`::=`	`(PeType PeType)`
`FcnType`	`::=`	`top-fcn` | `(Src Env)`
`Src`	`::=`	`<source language expressions>`
`Env`	`::=`	`<representations of` $Sym \rightarrow$ `PeType>`
`PeType`	`::=`	`NumType` | `BoolType` | `SymType` | `NilType` | `PairType` | `FcnType` | `top`

Figure 3: FUSE representation of types

adds "top" elements to the various subdomains (*i.e.,* "lifts" the CPOs), since each top element can be interpreted as describing all of the elements below it. Therefore, in addition to values, FUSE's approximations may also contain representations of $\top_{Num}$, $\top_{Bool}$, $\top_{Sym}$, $\top_{Str}$, $\top_{Func}$, and $\top_{Val}$. There is no need to represent $\top_{Nil}$ or $\top_{Pair}$, since *Nil* has only one element, and because the "least known" pair is represented as $< \top_{Val}, \top_{Val} >$. Functions are represented intensionally by their source text and environment. Generalizing two functions with the same source text generalizes their environments; other otherwise, $\top_{Func}$ is used. FUSE's repertoire of approximations is shown in Figure 3.

This scheme represents known values exactly, and preserves the structure of partially known values with known, finite structure. It cannot, however, describe the structure of infinite structured objects (such as the set of all lists of integers), describe disjoint unions (the set $\{1.2, 3, 5\}$), or describe arbitrary constraints (such as the set of all pairs of numbers whose sum is 3, or the set of values that are not pairs).

Generalization of these types is accomplished in the obvious way: by climbing the lattice until a least upper bound is found.

1.3.2 Symbolic Values

FUSE's process for computing a residual program can be considered as interpreting the source program under a nonstandard semantics [15] (which performs generalization, caches specializations, and leaves behind a trace of residual code) and a nonstandard value domain (which can represent unknown values, etc). FUSE's value domain is made up of *symbolic values*, which are described in detail in [45]. A symbolic value has several attributes, one of which is the type approximation drawn from the type system of Figure 3, denoting the values that that could possibly appear in place of the symbolic value at runtime. Another important attribute contains a residual code expression which can be used to compute the symbolic value's value at runtime. After specialization is complete, a separate code generator walks the graph formed by the code attributes and produces a residual Scheme program. Several other attributes are used for other purposes, but will not be described here; for more details, see [35].

1.3.3 Termination

FUSE achieves termination through *pairwise generalization*, as described in [45, 36] (this termination strategy is also used by [37, 43]). That is, the specializer maintains a model of the runtime stack, and uses it to eliminate unbounded recursion by generalizing the argument vectors of recursive calls which it cannot prove are well-founded (actually, a combination of heuristic methods, machine-generated annotations, and human-generated annotations are used).

The exact mechanism is not important: what matters is that the specializer have *some* mechanism that is responsible for building a finite number of specializations (limiting specialization by generalizing arguments), each of which is of finite size (limiting unfolding by building specializations). We will make use of this later when arguing that our information preservation mechanisms terminate (*c.f.* Section 3.4.2 and Section 4.4.1.3). The basic idea will be that the information preservation mechanisms serve only to introduce better approximations, and will do so in finite time and space—preventing nontermination due to the `use` of such approximations in performing reductions is the responsibility of the termination mechanism.

2 Two 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.

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, is an extension of the by-now-canonical "MP" language, first used as an example by Sestoft [39]. 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 5 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 (`cdrs`) of bindings in this store are altered as interpretation proceeds, but the names (`cars`) never 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`

```
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: Syntax of the MP+ language

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 6. 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. 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 <any value> instead of ((x . <any value>) (y . <any value>)), meaning that any accesses to the store after the program exits the while loop (*e.g.,* procedure lookup6 and update5 in Figure 7) 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 'foo y)) are residualized in this manner, but in general, all store accesses after a loop exit will be residualized sub-optimally.

```
(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 5: Fragment of Interpreter for MP+

```
(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 6: 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.

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 a `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 4.6), specialization of the interpreter on MP+ programs which use procedures (*e.g.,* contain `call` statements) can cause the construction of truly recursive loops, whose return value *is* used by their bodies.

There are three common work-arounds for the problem illustrated by this example, all of which use syntactic transformations to avoid passing the names upward. The first solution 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, 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. For programs intended as input for offline specializers, this transformation can be automated [31, 16]; these transformations are too conservative for input to an online specializer because only values that are provably "static" will be factored out into downward-passed lists, and preserved during specialization.

Another solution is rewrites the program so that it only passes information downwards, never upwards. This approach is taken by Mogensen [31] and by Launchbury [29], whose interpreters pass an extra argument containing the "rest" of the MP+ statements to be executed. Instead of

```
(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 7: 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.

```
(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 8: Divide-and-Conquer integration program

returning a store when the loop is complete, `mp-while` exits by calling `mp-command` on the "next statement" and the store. This approach involves considerable hand-rewriting, and has not, as yet, been automated.

The third common work-around performs CPS [42] transformation on the program, as is done by Consel and Danvy [14]. 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. However, CPS does have a cost: it introduces higher-order constructs which complicate specialization, and may reduce polyvariance. Furthermore, as described in [36], for truly recursive programs, this approach trades one information loss problem for another; we will cover this in Section 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 8). 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, and sums them.[4]

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 9). 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.

[4] 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 9: 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 10: 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, `+`.

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 10).

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) 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.

2.3 Commentary

In both of these examples, achieving a good specialization required that the specializer compute accurate approximations to values returned by residual calls to specializations. Thus, both examples serve to motivate the mechanism that will be described in Section 3.3; we will revisit these examples and demonstrate that mechanism in action in Section 3.5.

As we shall see in Section 4, the need for accurate return value approximations can be avoided entirely by performing the CPS transformation [42, 14, 36] on the program to be specialized. However, for these examples, this transformation only serves to trade one information loss problem (approximations to return values) for another (approximations to parameter values). Thus, in CPS-converted form, these examples serve to motivate the mechanism of Section 4.4, which computes accurate approximations for values passed as parameters to specializations of higher-order procedures. That mechanism will be demonstrated in Section 4.6, in which we will revisit both of these examples in CPS-converted form.

3 First-Order Programs

In this section, we treat an instance of unnecessary information loss that occurs in existing specializers for first-order programs, namely, the use of "any value" as the approximation to values returned by calls to specializations.[5] We describe an algorithm for computing more accurate approximations to return values, its implementation in FUSE, and its use in specializing realistic programs.

3.1 Sources of Information Loss

As we described in Section 1.2, a program specializer inherently loses information since it uses a finite type system to represent a possibly infinite collection of runtime values. It loses further information when it generalizes two approximations; in online specializers, such generalization takes place both when computing the return value of an `if` expression with unknown test [15], and when computing the argument specification to be used in building a specialization [45]. These losses are inherent in the nature of specialization itself, but can be mitigated to some extent through the use of a more precise type system.

However, information loss in existing specializers is not limited to these cases. In particular, existing specialization techniques for first-order programs always use "any value" 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[6]

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

is specialized on `x=<any value>`, 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 "any integer" still do not achieve this result, because the recursive call to `length` is viewed as returning "any value."

3.2 Existing Approaches

Existing approaches to dealing with the problem of computing approximations of return values of residual procedure calls concentrate on *avoiding* the problem rather than 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

[5]Actually, the argument we will make is true for all programs which, when specialized, result in first-order residual programs (*i.e.,* as long as all of the higher-order functions are inlined, we don't have to treat them specially when computing approximations, but can use the same mechanisms we use for first-order code), but this is a difficult class of programs to characterize, as it depends not only on the program, but on the specializer, and on the input approximations given to the specializer. To avoid such characterizations, we will deal exclusively with first-order programs in this section.

[6]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 2.1).

described in Section 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 was fully general and automatable, but since it builds higher-order programs, it cannot be considered a solution for first-order specializers; we will address the merits of CPS when we describe information loss in higher-order specializers in Section 4.

3.3 Fixpoint Iteration Solution

Instead of rewriting the source program to avoid the consequences of overly general approximations to return values, we can 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.

Consider the recursive `length` example again:

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

We would like to compute an approximation to the return value of the expression `(length y)` where `y=<any value>`. 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 x))`'s return value, we can see that computing the return value of `length y` requires computing the return value of `(length (cdr x))`. In the residual program, however, both the initial and recursive calls to `length` will invoke the same specialization (because both calls bind `x` to the same type, namely `<any value>`). Thus, the return value of the specialization depends on itself.

This suggests a fixpoint solution, in which the return value of a residual procedure is computed repeatedly until a least upper bound is found. This can be accomplished by adding a unique bottom element to the specializer's type lattice (we'll call it `<no value>`) and using it as the initial approximation to the return value of a residual procedure call. This 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 Abstract Interpretation frameworks such as Binding Time Analysis [26, 31]. Our solution bears an even greater similarity to the Minimal Function Graph (MFG) framework of [24]; 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 just MFG fixpointing over the cache.

In order to be sure that the fixpoint iteration terminates with a correct least upper bound, the specializer must guarantee that (1) the type lattice has a finite height, and (2) all type operations in the specializer are monotonic. These are restrictions on the type system of the specializer; we will examine them in the context of FUSE in Section 3.4.2.

Assuming, for the moment, that these restrictions are met, let us consider the specialization of the `length` example. In Figure 11, rows represent source expressions, while columns represent

Expression	Iteration 1	Iteration 2	Iteration 3
`x`	`<any value>`	`<any value>`	`<any value>`
`(null? x)`	`<any value>`	`<any value>`	`<any value>`
`(cdr x)`	`<any value>`	`<any value>`	`<any value>`
`(length (cdr x))`	`<no value>`	`0`	`<any integer>`
`(+ 1 (length (cdr x)))`	`<no value>`	`1`	`<any integer>`
`(if (null? x) 0 (+ 1 (length (cdr x))))`	`0`	`<any integer>`	`<any integer>`
`(length y)`	`0`	`<any integer>`	`<any integer>`

Figure 11: Fixpoint Iteration on `length` example

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.

3.4 Implementation in FUSE

3.4.1 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 `no-value` 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.

3.4.2 Termination

As we saw above, 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 specializer's operations on types, 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.[7]

We state, without proof, that all of the specializer's type operations (including type computations in primitive operators, special forms, and the generalizer used by the termination mechanism), are monotonic. Were this not so, the approximations computed by the iteration could oscillate between higher and lower lattice values forever.

The finite height requirement is needed to avoid 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

{`<no value>`, `(1 . <no value>)`, `(1 . (1 . <no value>))`, ...}.

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 functions (because their environments can contain pairs and functions). Functions are not an issue for finiteness; because our program is first order, their number is fixed by the source program, they are never passed as arguments, or returned, and they are never generalized. Thus, we only need to avoid infinite ascending chains of pairs.[8]

The easiest way to avoid infinite ascending chains is by construction: all we have to do is prevent the bottom element `<no value>` from appearing in any pair. That is, `cons` applied to `1` and `<no value>` returns `<no value>` instead of `(1 . <no value>)`. This guarantees that, after the first iteration, the return value approximation is either `<no value>`, in which case the specializer has proven that the specialization doesn't terminate, and can return `<no value>` with a clear conscience, or it contains no occurrences of `<no value>` 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 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

[7]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.

[8]The algorithm described in Section 4.5, which does handle functions, is not vulnerable to infinite ascending chains.

pairs; *e.g.,* `(1 . (1 . <no value>))` dominates `(1 . <no value>))`. 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 [35] 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.

3.4.3 Technicalities

In this section, we cover a few implementation technicalities. First, we note that fixed point iteration necessitates a change to the control structure of FUSE as described in Section 1.3 and in [45]. 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 `<any value>`. During the first iteration of the fixed point computation for this specialization, `b` will also be specialized on `<any value>`. 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. 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 bottom as the approximation to the specialization's return value, but may proceed from the previous approximation.

Second, we address the cost of this mechanism. 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 go 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 [31], 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 [30] describes the use of lazy techniques to reduce representational overhead, but that's a different problem). In Section 4.5, 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,[9] and (2) because the recomputations, starting higher in the lattice, often converge faster than the original computations did.

Another, more serious cost, is that fixpoint iteration over nested recursive procedures may interact badly with FUSE's specialization cache, causing unnecessary specialization. The reason is that, each time the outer loop's approximation is recomputed, a new procedure object representing the inner loop is constructed. For various technical reasons having to do with the lexical addressability of free variables, the cache of specializations is indexed by the identity of these objects, rather than by their intensional equality; this means that a new specialization (and return approximation) is constructed for the inner loop for each fixpoint iteration of the outer loop. Similarly, the improved caching scheme of [35] cannot be used here, as it only considers procedure arguments, not free variables, when determining when specializations can be re-used. We solve these problems by disallowing nested procedures, transforming such programs to a "flat" form via a process similar to lambda-lifting [23]. In this form, inner loops take their free variables as arguments, making them visible to the specialization cache and to the mechanism of [35].

3.5 Examples

In this section, we demonstrate our fixpoint iteration mechanism on the examples of Section 2.

[9] 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.

3.5.1 Interpreter Example

Consider specializing the interpreter of Figure 5 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))))
```

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 . <any value>) (y . <any value>))`, 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 `<no value>`. Thus, the approximation for the `if`, and thus the procedure body, is the generalization (least upper bound) of the approximation to the store and `<no value>`, which is `((x . <any value>) (y . <any value>))`. 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 . <any value>) (y . <any value>))`, allowing the calls to `lookup` and `update` to be unfolded. The final code is that shown in Figure 6.

3.5.2 Integration Example

The integration code shown in Figure 8 is a higher-order program, since the function to be integrated, `fcn`, is passed as a parameter. Because this section addresses only first-order programs, we have to change the example slightly.[10] We will remove the formal parameter `fcn` and instead add an explicit definition binding `fcn` to a function, yielding the source program shown in Figure 12.

[10] In actuality, there is no problem as long as `fcn` is known at specialization time, since it will just be unfolded, and analyzed just as though the function definition was present inline in the source program. However, we are trying to avoid the first-order-residual characterization of programs, and are therefore presenting a truly first-order example.

```
(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 12: First-order Divide-and-Conquer integration program

Consider specializing this code on `lhs` and `rhs` both specified as "any 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, `<no value>`; their sum gets the same approximation. The approximation for the `if` is the generalization of `<any number>` and `<no value>`, which is `<any number>`; 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 to be specialized on numbers, and return `<any number>`. The `if` expression, and thus the body of the specialization, now returns `<any number>`; since the approximation is unchanged, the fixpoint iteration terminates. The resulting residual program is identical to that shown in Figure 9.

4 Higher-Order Programs

The treatment of information recovery in Section 3 and in [46] 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 by examining what it means to specialize a higher-order program, and show that the usual approach to specializing higher-order procedures (as used in [45, 4, 12]) 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. We describe an algorithm for computing more accurate approximations to parameter values, its implementation in FUSE, and its use in specializing realistic programs.

4.1 Basics

Higher-order programs require that the program specializer be able to operate on first-class functions[11], 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 [4]) or by nesting specializations (as in the FUSE code generator [45]). As with first-order procedures, this unfolding/specialization process is polyvariant.

This takes care of many higher-order uses of functions, such as `(map (lambda (x) ...) l)`, where inlining the `lambda` expression is sufficient. Similarly, handling CPS-transformed versions of tail-recursive procedures is simple, because the continuation[12] is always inlinable. For example, specializing

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

on `k=(lambda (x) (+ x 5))`, and `n` and `ans` unknown, might yield

```
(define (fact43 n ans)
  (if (= n 0)
      (+ ans 5)
      (fact43 (- n 1) (* n ans))))
```

Difficulties arise when a call head evaluates to an unknown value at specialization time; *i.e.,* when the residual program is higher-order. This occurs when closures from multiple `lambda` expressions reach a single call site, as in

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

in which both invocations of `k` are reached both by the initial continuation to `fact`, and the recursive continuation `(lambda (ans) (k (* ans)))`. This situation forces the specializer to residualize the construction of (*i.e.,* build specializations of) all closures which might reach this site at runtime. Unlike the case of first-order procedures, in which the specializer can residualize a call simply by

[11] We assume, for simplicity's sake, that first-order and first-class procedures are distinguished syntactically: first-order procedures are built by `define`, 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 [4] and Schism [12]. Our approach does not rely on any such distinction; we make it only to simplify the discussion.

[12] In this paper, the term "continuation" refers to a first-class procedure introduced by the CPS transformation, not to any implicit continuation present in the Scheme evaluator or reified by `call-with-current-continuation`.

building a call to the appropriate specialization, both residual call sites of `k` must be able to invoke either of `two` specializations at runtime, depending on which `lambda` expression the call head came from. 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[13], 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.[14] 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 × `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.[15] Once first-class environments become part of the Scheme language, this approach may become more appropriate. Schooler's IBL [38] 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.

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 higher-order `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

[13]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.

[14]A practical implementation of this might build a first-order 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.

[15]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.

the best of our knowledge, this approach has not been implemented,

A third approach allows the building of only a single residual version of each specialization-time closure object (in cases where unfolding or specialization of first-order procedures duplicates `lambda` expressions, each copy may induce a separate specialization.). That specialization must be sufficiently general to be valid at all of its call sites. Existing specializers provide such applicability by building all first-class specializations on completely unknown arguments. As we shall see, this approach builds specializations which are overly general since the call sites may have some amount of known information in common, which could be used in building the specialization. At first, this might seem to be a completely *monovariant* solution, because only one specialization is permitted. In actuality, 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. As described above, this handles many "packaging" uses of first-class functions, such as passing a closure to `map`. Second, first-order 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 [45], Similix-2 [4], and Schism [12], and is the approach we will consider in the remainder of this section.

Note that the second two approaches, because they can generate multiple residual `lambda` expressions from a single `lambda` expression in the source, can break code that uses `eq?` on closures (the first approach doesn't have this problem because `eq?` on closures can be simulated via the use of `eq?` on environments). We find the use of `eq?` on closures questionable, and have not concerned ourselves with this problem.

4.2 Sources of Information Loss

As with first-order programs, a specializer will lose information as it processes a higher-order program. These losses stem from the use of a finite type system, and from the need to generalize approximations when computing the return values of `if` expressions and the actual parameters to use when building a specialization. We saw above that existing specializers lost information unnecessarily when computing the approximation to the value returned from a residual procedure call.

In the case of higher-order programs, the need to compute approximations to return values can be eliminated by transforming the program into continuation-passing style (CPS) [42] form. In CPS programs, all calls are tail calls in the sense that no reductions are performed on their return value; 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 residual 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 paper, 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 [14], who used it to improve the accuracy of an offline program specializer that lacks 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 (and, in fact, any offline specializer; see [36] for details), which cannot perform generalization, the return value of this fragment is "any value." The best result possible from a specializer with generalization would be "a pair, whose car is either 1 or 2, and whose cdr is any value." This solution is better than "any value," 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[16] 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 "a pair whose car is 1 and whose cdr is any value," while the other will be unfolded on "a pair whose car is 2 and whose cdr is any value." 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 [36]. However, experiments to date [14, 28] 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 x ans)
  (if (null? x)
      ans
      (tail-length (cdr x) (+ 1 ans))))
```

specialized on x unknown and ans an unknown integer[17], returns "any integer" under specializers which compute return values, but only "any value" under specializers which don't. If we transform it to

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

[16]This is not the full CPS transform of [42], 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.

[17]A user would most likely specialize this procedure on ans=0, but termination criteria (either manual or automatic) will raise ans's approximation to "any integer" in order to build a finite number of specializations of length.

we no longer have this problem. When we specialize this transformed version on `x` unknown, `ans` an unknown integer, and `k=(lambda (loop-result) ... loop-result ...)`, `k` will be applied to (and unfolded on) the approximation "any 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 4.6, this means that the CPS transformed version of the MP+ interpreter, specialized on the example program of Section 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 3.1. After CPS transformation, it looks like

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

When we specialize this on unknown `x` and `k=(lambda (result) ... result ...)`, the specializer will build specializations of both the initial continuation bound to `k`, and the continuation `(lambda (ans) ...)` passed on the recursive call. Since both residual invocations of `k` will pass an integer argument, it would be desirable to specialize the continuations on "any integer," which would allow the expression `(+ 1 ans)` in the recursive continuation to be simplified to `(integer+ 1 ans)`. If this were done, the specialized program would be be as good as the first order fixpoint iteration solution. Computing this approximation requires that the specializer find all call sites of each continuation (in this case, both calls to `k`), and compute an approximation that denotes all values which could be passed in at any of those sites (in this case, "any integer" includes both the `0` passed at the site `(k 0)` and the "any integer" passed at the site `(k (+ 1 ans))`).

Unfortunately, existing specializers for higher-order programs always use "any value" as the approximation to the value(s) passed to residual `lambda`-expressions, even when their type system might be able to represent a better approximation to that value(s). This unnecessary loss of information means that residual `lambda` expressions constructed by such specializers are unnecessarily general. In particular, with respect to the computation of accurate return value approximations, it limits the utility of the CPS transformation to programs whose residualization is tail recursive. For example, when the `length` program above is specialized, neither continuation can be unfolded; instead, both continuations are specialized on a completely unknown actual parameter, forcing the `+` in `(+ 1 ans)` to be left residual in its most general form. Similarly (*c.f.* Section 4.6), 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.

4.3 Existing Approaches

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[18], 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 [36].

4.4 Control Flow Analysis Solution

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:

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, 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 will consider each of these issues in turn.

4.4.1 An Accurate Specialization Algorithm

4.4.1.1 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 13) 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 $\perp$, and leave the body empty.

[18] 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.

```
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(S.ARGS) for all S in SITES(L)
    If A=L.ARGS then
       exit
    else
       set L.ARGS:=A
       set L.BODY:=specialize(L.CLOSURE,L.ARGS)
       goto LOOP
```

Figure 13: An iterative specialization algorithm

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`).[19] For each `lambda` expression, compute the least upper bound of the argument approximations at all of its call sites. If this approximation is the same as the expression's argument vector, do nothing.[20] 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.

4.4.1.2 An Example

Consider the `length` function, specialized on a known continuation `k=(lambda (result) (+ 5 result))` and an unknown list `x=`$\top$. This process is shown in Figures 14 and 15.

On the first iteration of the algorithm, the initial and recursive continuations are specialized on $\perp$, yielding residual `lambda` expressions with empty bodies. Control flow analysis finds one

[19] If closures constructed from the `lambda` expression can be returned out of the top-level invocation of 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 [41].

[20] 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 ⊥
                <empty>)
              (cdr x))))

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

SITES((lambda (ans) ...))    = {(k 0)}
SITES((lambda (result) ...)) = {(k 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)}
SITES((lambda (result) ...)) = {(k 0), (k 1)}
```

Figure 14: 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)}
SITES((lambda (result) ...)) = {(k 0),
                                (k (integer+ 1 ans)}
```

Figure 15: Applying the iterative algorithm to the `length` program (continued)

call site for each residual `lambda` expression, namely `(k 0)`. Since $0 \neq \bot$, 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)`. Computing the least upper bound of `0` and `1` gives $\top_{integer}$, which is not equal to the previous approximation, `0`. Respecialization of both continuations on $\top_{integer}$ produces 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 of the argument approximations at each call site ($0 \sqcup \top_{integer} = \top_{integer}$) is the same as 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 $\top$, the applications of `+` in those continuations would not be specialized to `integer+`.

4.4.1.3 Termination and Correctness

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 first-order 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 [4, 45].

The correctness of this algorithm can be shown 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.

4.4.2 Control Flow Analysis

As it stands, the algorithm of Section 4.4.1.1 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.

4.4.2.1 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 [41] and Harrison [20], while simpler, less accurate solutions are used by Sestoft's "closure analysis" [40], Bondorf's variant of this analysis [4], and Consel's higher-order binding time analysis [12].

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 proposes a taxonomy of analyses in terms of the depth of the call history used to distinguish between control paths (*i.e.,* "0CFA" maintains one set of abstract closures[21], "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, etc). Analyses higher in the taxonomy compute more accurate estimates, but are more costly to compute [27].

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

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

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

[21] 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.

0CFA will determine that the call site (k x) could invoke either (lambda (a) ...)[22] 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 $\top$, 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.

First, any calls 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; no residual higher-order code is generated, 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.

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=<any integer>, and y=<any list>. 0CFA on the source program would determine that either continuation could be invoked on either an integer or a list. In order 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."[23] 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

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

[23] 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 can be counterproductive.

residual program, it will notice that the continuation (lambda (a) ...) is called only from the specialization with x=<any integer>, and the continuation (lambda (b) ...) is called only from the specialization with x=<any list>, and will compute the approximations we 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 ($\top$). 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 (possibly invalid [35]) 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.

4.4.2.2 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 14), 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 lambdas reach all calls 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 see this, consider running our algorithm using an accurate control flow analysis. For a particular specialization, the set of call sites found by successive applications of the control flow analysis does not increase monotonically. However, we are not interested in the set of call sites *per se*, but rather in the least upper bound of the argument approximations at those call sites. This upper bound does increase monotonically, even when the set of call sites does not. Thus, retaining call sites from prior applications of CFA would not affect the upper bound.

Consider a residual `lambda` expression with call sites c_1 and c_2, with argument vectors v_1 and v_2, respectively. Suppose that our algorithm respecializes the `lambda` expression on $v_1 \sqcup v_2$, yielding call sites c_1 and c_3, with argument approximations v_1 and v_3. The new argument vector for further respecialization is $v = v_1 \sqcup v_3$. If an inaccurate CFA algorithm were to keep the old call site c_2, we would instead compute $v' = v_1 \sqcup v_2 \sqcup v_3$ as the new argument vector. But we already know that $v_1 \sqcup v_3 \sqsupseteq v_1 \sqcup v_2$, so $v' = v$. Retaining the old call site c_2, which no longer appears in the residual program, does not affect the result. Thus, 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.

4.5 Implementation in FUSE

This section describes the implementation of control flow analysis and higher-order specialization in FUSE [45]. 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 [14, 36], but also allows us to simplify the control flow analysis.

4.5.1 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).

4.5.2 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.

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=`$\top$. The specializer first builds 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 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[24] 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)))
```

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.

4.5.3 Technicalities

In this section, we cover a few implementation technicalities. First, the description above stated that, when a new specialization of a closure is constructed, it is stored in the closure object. 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, 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.

Second, we address the cost of this mechanism. 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.

[24] 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.

More importantly, this mechanism is more efficient than the first-order solution of Section 3.3. 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 3.4.3). This is particularly troublesome in the presence of tail-recursive loops, where the return value approximation is never used during 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.

4.6 Examples

In this section, we demonstrate our mechanism on CPS-converted forms of the examples of Section 2.

4.6.1 Interpreter Example

Consider the CPS converted form of the interpreter in Figure 5. 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 16. 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 respecialization 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 16: 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 17: 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.

```
(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 18: 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.

```
(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 17. 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 . <any value>) (b . <any value>) (out . <any value>))` to the store passed to this continuation. If traditional methods had been used, the continuation would have been specialized on the approximation `<any value>`, and all store accesses in the continuation would have been residualized as loops, rather than as open-coded `car` and `cdr` instructions.

4.6.2 Integration Example

We now return to the integration program shown in Figure 8. We would like to show how our mechanism specializes the CPS transformed version of this program. If we specialize the CPS-transformed integration program on an unknown continuation, `fcn=(lambda (x) (* x x))`, and `lhs` and `rhs` known to be numbers, we get the specialization shown in Figure 18. 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 3.5.2, 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 `<any number>`.

5 Related Work

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

5.1 Specializers

Although no existing specializer performs fixpoint iteration to compute return value approximations, or uses approximations other than "any value" for the values of formal parameters when building specializations of higher-order procedures, a variety of techniques have been used to improve the quality of specialization.

Haraldsson's REDFUN-2 [19], 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 higher-order functions. REDFUN-2 only automatically derives type information from residual function calls in very restricted situations.

The online systems of Berlin [3] and Schooler [38] 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 [15] 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 higher-order 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 [31] and Consel [11], the projection-based analysis of Launchbury [29], and the higher-order binding time analyses of Bondorf [4], Mogensen [31], and Consel [12] reason about structured and function types, allowing offline specializers to make use of more information than the simple scalar BTA of MIX [26]. 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 higher-order procedures has not been implemented; based on the analysis of [36], we believe it will be difficult to perform these optimizations in an offline framework.

5.2 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 [5]). Automating such replication was first suggested by Mogensen [31]. Using the CPS transformation to perform the same task was first suggested by Consel and Danvy [14] and has been used to improve the quality of specialization in several examples [13, 28]. other manual transformations, such as eta-conversion, are also common; Holst and Hughes [21] suggest a possible means of automating such transformations.

5.3 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 3.3 is similar to the Minimal Function Graph analysis presented in [24], 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 [31], we believe that ours is the first application to the specialization phase.

Young and O'Keefe's *type evaluator* [47] 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 [32, 2] 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.

5.4 Control Flow Analysis

Control flow analysis for Scheme programs has been investigated extensively in the context of parallelization by Harrison [20] and in the context of program optimization by Shivers [41]. Existing applications of control flow analysis to program specialization are Bondorf's application [4] of Sestoft's closure analysis [40] and Consel's higher-order binding time analysis [12]. Both of these analyses operate on source programs prior to specialization; we believe ours is the first application of CFA technology during specialization.

5.5 Compilers

The optimizing compiler technology of Chambers and Ungar [10] 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 [45, 37, 43]. 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.

The polymorphic inline caching methods of Hölzle [22] suggest a possible new frontier for program specialization. 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.

Conclusion

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 and formal parameters of specializations of higher order functions. The overly general approximations presently used in such cases adversely affect the quality of residual programs.

We have presented algorithms for computing better approximations in both cases: a fixpoint iteration method for computing return value approximations and a control flow analysis method for computing formal parameter approximations, and have shown how these algorithms are implemented in our specializer, FUSE. Adding these methods to FUSE have allowed it to build better specializations of several real-world programs.

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 and control flow analysis prior to specialization time would establish conservative upper bounds on return value approximations and control flow information. Such bounds would allow the specializer to halt fixpoint iteration or stop propagating initial source/final destination information when it detects that the bound has been reached, avoiding unnecessary work. Finally, we plan to explore interactions between these information preservation mechanisms and the "re-use" mechanism of [35].

Acknowledgements

The first author would like to thank Jim des Rivieres, John Lamping, and Carolyn Talcott for their comments on drafts of this paper.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] 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.

[3] 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.

[4] 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.

[5] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[6] 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.

[7] 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.

[8] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[9] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[10] 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.

[11] 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).

[12] 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.

[13] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

[14] 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.

[15] 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.

[16] 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.

[17] C. Gomard and N. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] A. Kanamori and D. Weise. An empirical study of an abstract interpretation of Scheme programs. Unpublished manuscript, 1991.

[28] 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.

[29] J. Launchbury. *Projection Analysis of Functional Programs.* PhD thesis, Glasgow University, 1991. to be published.

[30] 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.

[31] T. Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhangen, Copenhagen, Denmark, March 1989.

[32] B. R. Murphy. A type inference system for FL. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[33] J. Rees, W. Clinger, et al. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[34] 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.

[35] 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.

[36] E. Ruf and D. Weise. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[37] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, March 1991. Report TRITA-TCS-9101, 170 pages.

[38] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, MIT, Cambridge, MA, August 1984. Published as MIT/LCS/TR-324.

[39] 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.

[40] P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, 1988. Published as DIKU Student Report 88-7-2.

[41] 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.

[42] G. L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1978.

[43] 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.

[44] D. Weise. Graphs as an intermediate representation for partial evaluation. Technical Report CSL-TR-90-421, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990.

[45] 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.

[46] 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.

[47] 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.