Improving the Accuracy of Higher-Order Specialization using Control Flow Analysis*
FUSE MEMO 92-11

Erik Ruf†
Daniel Weise‡
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-2140
{ruf,weise}@cs.stanford.edu

Abstract

We have developed a new technique for computing the argument vectors used to build specializations of first-class functions. Instead of building these specializations on completely dynamic actual parameters, our technique performs a control flow analysis of the residual program as it is constructed during specialization, and uses the results of this analysis to compute more accurate actual parameter values. As implemented in the program specializer FUSE, our technique has proven useful in improving the specialization of several realistic programs taken from the domains of interpreters and scientific computation. Also, it extends the utility of the continuation-passing-style (CPS) transformation for binding time improvement to programs with non tail-recursive residual loops.

Introduction

The treatment of function calls in program point specializers for first-order functional languages is fairly straightforward: since the head of the call always evaluates to a known procedure at specialization time, the specializer is free to either reduce the call, replacing it with the result of unfolding the procedure's body on its argument, or to residualize the call, replacing it with a call to a specialized procedure. Virtually all existing specializers are *polyvariant*, meaning that the specializer is free (modulo termination issues) to unfold or specialize each procedure an arbitrary number of times on arbitrary argument vectors.

Adding first-class procedures to the language complicates things somewhat. At specialization time, call heads may now evaluate to specialization-time closures or to completely dynamic (unknown) values, in addition to first-order procedures.[1] 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 dynamic 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 [1]) or by nesting specializations (as in the FUSE code generator [14]). As with first-order procedures, this unfolding/specialization process is polyvariant.

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

Figure 1: A continuation-passing style length function

Things become interesting when a call head evaluates to a dynamic value at specialization time; *i.e.,* when the residual program is higher-order. This situation forces the specializer to residualize the construction of (*i.e.,* build specializations of) all closures which might reach this site at runtime. Both existing specializers and the technique described in this paper build such specializations in a *monovariant* manner; each specialization-time closure used in a first-class manner is represented by a single `lambda` expression in the residual program.[2] As a consequence of monovariance, if a single closure might reach several such sites, its specialization must be sufficiently general to be applicable at all of them.

Existing specializers provide such applicability by building all first-class specializations on completely dynamic arguments. This approach builds specializations which are overly general since the call sites may have some amount of static information in common, which is not used in building the specialization. For example, consider the CPS-converted form of a recursive `length` function (Figure 1) specialized on static `k` and dynamic `lst`. Current approaches will specialize both the initial continuation[3] (bound to `k`) and the recursive continuation (`(lambda (ans) ...)`) on a dynamic

*Initially published in *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pp. 67-74, San Francisco, June, 1992. These proceedings are available as Yale Technical Report YALEU/DCS/RR-909.

†Supported by an AT&T Bell Laboratories Ph.D. Scholarship.

‡Supported in part by NSF Contract No. MIP-8902764, and in part by Defense Advanced Research Projects Agency Contract No. N0039-91-K-0138.

[1] 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 [1] and Schism [2]. Our approach does not rely on any such distinction; we make it only to simplify the discussion.

[2] Two polyvariant alternatives, and our rationale for remaining with a monovariant strategy, are discussed in [9].

[3] 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**.

actual parameter, forcing the `+` in `(+ 1 ans)` to be left residual in its most general form. Since both residual invocations of `k` will pass an integer argument, it would be preferable to specialize the continuations on "any integer," which would allow the expression `(+ 1 ans)` in the recursive continuation to be simplified to `(integer+ 1 ans)`. Our goal is to achieve this specialization.

To accomplish this, we must compute, for each first-class specialization (residual `lambda` expression), 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 Figure 1, the call to `k` inside the recursive continuation must be discovered before that continuation is specialized.).

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

This paper has four sections. The first describes an iterative algorithm for computing accurate specializations of first-class procedures, solving problem (1). Section 2 treats problem (2); we show that a somewhat inaccurate control flow analysis is sufficient for our purposes, and describe an incremental control flow analysis (CFA) algorithm which avoids the need to re-analyze the entire program when only a small portion of it changes. We then give several examples of our approach (Section 3) and conclude with a discussion of related work in program specialization and control flow analysis (Section 4).

1 An Accurate Specialization Algorithm

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

```
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 2: 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`).[4] 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.[5] 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.

1.2 An Example

Consider the `length` function of Figure 1, specialized on a static continuation `k=(lambda (result) (+ 5 result))` and a dynamic list `lst=`$\top$. This process is shown in Figure 3.

On the first iteration of the algorithm, the initial and recursive continuations are specialized on $\perp$, yielding residual

[4] 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 dynamic 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 [12].

[5] 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)}
```

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 3: Applying the iterative algorithm to the `length` program

`lambda` expressions with empty bodies. Control flow analysis finds one call site for each residual `lambda` expression, namely `(k 0)`. Since $0 \neq \bot$, both continuations are respecialized 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+`.

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 [1, 14].

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.

2 Control Flow Analysis

As it stands, the algorithm of Section 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.

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 [12] and Harrison [6], while simpler, less accurate solutions are used by Sestoft's "closure analysis" [11], Bondorf's variant of this analysis [1], and Consel's higher-order binding time analysis [2].

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[6], "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 [7].

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))
```

0CFA will determine that the call site `(k x)` could invoke either `(lambda (a) ...)`[7] 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 `call`s for which the specializer can prove that the head is reached only by closures generated by a single `lambda` expression are either unfolded or specialized; 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.

```
(define (sum k x)
  (if (= x 0)
      (k 0)
      (sum (lambda (ans) (k (+ x ans)))
           (- x 1))))
```

Figure 4: Sum

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 a function `sum` that computes the sum of the integers from `0` to `x` (Figure 4), and two calls to `sum`:

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

where `y`=$\top_{integer}$, and `z`=$\top$. 0CFA on the source program would determine that either continuation could be invoked on either an integer or on any value whatsoever. However, the residual program will contain two specializations for `sum`, one with `x`=$\top_{integer}$ and one with `x`=$\top$; when 0CFA is run on the residual program, it will notice that the continuation `(lambda (a) ...)` is called only from the specialization with `x`=$\top_{integer}$, and the continuation `(lambda (b) ...)` is called only from the specialization with `x`=$\top$, and will compute the approximations we want. If both call sites of `sum` had passed an integer `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 ($\top$).

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 [8]) 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.

[6] 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.

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

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

2.2.1 Observations

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

Second, and, for our purposes, more interestingly, *the accuracy loss does not affect the quality of specialization.* To 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.

2.2.2 Implementation

This section describes the implementation of control flow analysis and higher-order specialization in a variant of FUSE [14], an online program specializer for a side-effect-free subset of Scheme. We assume that all user procedures and procedure applications in the input program have been CPS-converted;[8] this not only allows us to get good binding times without the need to compute return value approximations [3, 9], but also allows us to simplify the control flow analysis.

As described in [14], 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.

[8] This is not the full CPS transform of [13], 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.

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).

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

specialized on `k=(lambda (result) (+ 5 result))` and `lst=`$\top$. The specializer first builds a specialization of `length` on unknown `k` and `lst`:

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

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 `lst` has a final destination of `(cdr lst)`. 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 lst)
  (if (null? lst)
      (k 0)
      (length (lambda (ans) (k 1)) (cdr lst))))
```

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

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 lst2)
  (length (lambda (result) <empty>) lst2))
```

The construction of the residual invocation of `length` causes `lst2` to pick up the final destination of `lst`, namely `(cdr lst)`. 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[9] on $\top_{integer}$, yielding the final program

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

(define (length 2 lst2)
  (length (lambda (result) (integer+ 5 result)
                  lst2)))
```

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 `lst` and `(cdr lst)` were only derived

[9] 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.

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. 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.

3 Examples

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

In many programs, all of the specialization-time closures are unfolded. For example, many `lambda` expressions are used for implementing nonrecursive abstractions; a closure passed as the function argument to the `map` procedure will be unfolded as part of the unfolding/specialization of `map`. Similarly, in function language interpreters implementing environments with closures, if all environment lookups are resolvable at specialization time, no higher-order code will appear in the residual program [1].

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

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

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

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

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

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

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

Where our method shines is when first-class specializations *must* be constructed, even when CPS conversion is used. This occurs when not all continuation applications are unfoldable at specialization time; that is, when specializing a recursive procedure where a (non-unfoldable) recursive call passes a continuation different from that passed to the initial call. For direct-style programs, this is the case whenever the (non-CPS) residualization of the program is not tail recursive. The truly recursive length program of Figure 1 is one such example; usual specialization methods will specialize the recursive continuation on $\top$ rather than on $\top_{integer}$.

One might at first believe that the problem could be solved by type inference in a postpass or in the underlying Scheme compiler. Such an inference would deduce that `(lambda (ans) (k (+ 1 ans)))` is always called on an integer, and could replace the general `+` operator with `integer+`. This approach has two problems. First, if some other expression (such as a call to `integer?`) depends on the value of `ans`, it will not be reduced by the postpass/compiler, which is not a specializer, and cannot perform arbitrary reductions, construct specializations, etc. Second, a scalar type inferencer would not be able to optimize the program when ad hoc types are used (*i.e.,* determine that a closure should be specialized on the pair `(my-type-tag . ⊤)`).

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

Consider an interpreter for a small imperative language with global variables, `while` loops, and nullary procedure calls (*e.g.,* some dialect of BASIC). The interpreter maintains a store represented as an association list mapping identifiers to their values. When the interpreter is specialized on a static program and an dynamic input, the initial store maps the program's identifiers to dynamic values. It is imperative that the names in the store remain static throughout the specialization process so that all accesses and updates to the store can be resolved at specialization time, avoiding the need for linear search at runtime (and allowing for optimizations such as arity raising, which will remove the association list entirely). If the program contains a recursive loop of the form

```
(define-proc loop
  (if <exp1>
      (begin <cmd1>
             (call loop)
             <cmd2>)
      <cmd3>))
```

the residual program will contain a loop of the form

```
(define (eval-command cont store)
  (if <res-exp1>
      (eval-command
        (lambda (result-store)
          (cont <res-cmd2>))
        <res-cmd1>)
      (cont <res-cmd3>)))
```

where `<res-exp1>`, `<res-cmd1>`, `<res-cmd2>`, and `<res-cmd3>` are the residual code for `<exp1>`, `<cmd1>`, `<cmd2>`, and `<cmd3>`, respectively. If traditional methods are used, the specialization of the recursive continuation `(lambda (result-store) ...)` will be constructed on an argument value `result-store`$=\top$, and all store accesses/updates in `<res-cmd2>` will be residualized as calls to a search procedure. If our algorithm is used, the recursive continuation will instead be specialized on a store with static names, and all store accesses/updates will be residualized as open-coded `car` and `cdr` operations. These may then be converted into tuple accessors, or into separate identifiers, allowing constant-time access to values in the store.

Space considerations prevent us from including the text of the interpreter or the test program here; a complete version of this example appears in [9]. For a small program which computes the minimum of unary numbers represented as lists, specializing the interpreter using our algorithm reduced the execution time of the residual program by approximately 40% over the residual program produced by the FUSE of [14], which builds first-class specializations on $\top$. An additional 50% reduction was obtained when the arity raising enabled by our algorithm was performed.[10] Computing the specialization took 23% longer than under "vanilla" FUSE. Of the extra time required, 49% was spent recomputing one specialization, while the remainder of the time was spent in the incremental CFA algorithm.

4 Related Work

The existing specializers for higher-order untyped languages, Similix-2 [1], Schism [2], Lambda-Mix [4] and FUSE [14] ([5] and [10] treat higher-order languages, but cannot build first-class specializations), build a single specialization per dynamic `lambda` expression, using completely dynamic parameter values. Control flow analysis for Scheme programs is implemented in Sestoft's closure analysis [1, 11], Consel's higher-order BTA [2], and the analysis systems of Shivers [12] and Harrison [6]. The latter two analyses are capable of computing static information about parameters to first-class functions, but have not, as yet, been integrated with partial evaluators that can make use of such information. Our work can be viewed as an efficient integration of existing specialization and CFA techniques. Consel and Danvy [3] suggested the use of the CPS transformation for binding time improvement; their discussion was primarily motivated by dynamic `if` expressions and didn't treat the problem of specializing continuations bound in dynamically controlled non tail-recursive loops. Our work broadens the scope of programs for which their technique is beneficial.

Conclusion

Because existing specializers for higher-order languages use completely dynamic argument vectors when specializing first-class functions, they build overly general specializations. Our specialization algorithm computes more accurate argument vectors by using the results of a control flow analysis of the residual program, yielding better specializations. We do this efficiently by computing the control flow information incrementally.

Our incremental CFA technique may have applications outside of specialization. For example, polyvariant static analyses (such as BTA) for higher-order programs might encounter a "re-annotate, then recompute CFA" loop similar to the "respecialize, then recompute CFA" loop in our specializer.

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

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] E. Ruf and D. Weise. Preserving information during online partial evaluation. Technical Report CSL-TR-92-517, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

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

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

[12] 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.

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

[14] 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.