Opportunities for Online Partial Evaluation*

Erik Ruf and Daniel Weise

Technical Report: CSL-TR-92-516
(also FUSE Memo 92-7)

April, 1992

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

Abstract

Partial evaluators can be separated into two classes: *offline* specializers, which make all of their reduce/residualize decisions before specialization, and *online* specializers, which make such decisions during specialization. The choice of which method to use is driven by a tradeoff between the efficiency of the specializer and the quality of the residual programs that it produces. Existing research describes some of the inefficiencies of online specializers, and how these are avoided using offline methods, but fails to address the price paid in specialization quality. This paper motivates research in online specialization by describing two fundamental limitations of the offline approach, and explains why the online approach does not encounter the same difficulties.

Key Words and Phrases: Partial Evaluation, Program Specialization, Online Specialization, Offline Specialization, Binding Time Analysis, Binding Time Improvement.

*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 executing a program on the specification of its inputs. During this process, the specializer decides to reduce (*i.e.,* execute at specialization time) some expression, and to residualize (*i.e.,* put in the specialized program, for execution at runtime) others. These decisions affect the quality (both speed and size) of the specialized program, as well as the efficiency and termination properties of the specializer itself. In this paper, we examine two widely-used methods for making these decisions: the *offline* framework, in which certain decisions are made statically before the specializer runs, and the *online* framework, in which these decisions are made dynamically during specialization. We show that, in certain cases, the offline method delays computations which could be performed at specialization time until runtime, thus producing slower specialized programs than the online method, which can perform these reductions at specialization time.

This paper has four sections. The first defines program specialization; the second describes its offline and online variants. Section 3 analyzes several situations in which offline specializers have difficulties, and shows how these difficulties can be alleviated using an online strategy. Section 4 surveys related work on improving the accuracy[1] of specializers, both online and offline.

1 Program Specialization

In this section, we define program specialization, and its most common implementation, *polyvariant specialization* [8]. We will limit ourselves to describing the specialization of programs written in functional languages; our examples will be written in a functional subset of Scheme [33], occasionally extended with pattern-matching for brevity. Given this restriction, we will treat the terms "program" and "function definition" as synonyms. To denote the amount of information the specializer is given about a value, we use the terms "known" and "unknown," as well as "partially known."[2]

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

[1] We use the term "accuracy" of specialization to connote a measure of how many reductions a specializer is able to perform at specialization time. More accurate specializers perform more reductions, and produce faster residual programs.

[2] As we will describe in Section 2.1.1, the more commonly used terms, "static" and "dynamic" are not properties of values, but are instead properties of expressions, meaning "will denote only known values at specialization time," and "may denote unknown values at specialization time," respectively.

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 $E: \mathcal{L} \times V \rightarrow V$. 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 $SP: \mathcal{L} \times S \rightarrow \mathcal{L}$ 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 the original function; we consider reparameterization [39] 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 Polyvariant Specialization

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

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

Unfortunately, such a naive strategy often fails to terminate due to infinite unfolding of function calls in loops. Overly-eager reduction of procedure calls may miss opportunities for sharing in residual programs, and risks duplicating computations via beta-substitution. Finally, representational issues may forbid certain reductions, such as those which would place specialization-time data structures in residual contexts (see [5] for a discussion of some of these issues).[3]

Thus, all specializers limit specialization-time reductions by performing *folding* [9] operations. Folding is done by recursively specializing certain distinguished parts[4] of the program with respect to their arguments, and replacing their invocations with residual invocations of the corresponding specialization; this is often called *program point specialization* [26]. These specializations are cached in the specializer, so that they can be invoked from multiple call sites in the residual program; this allows for code sharing and also acts as a form of loop detection. Strategies for building and caching specializations vary. Most specializers use a strategy called *polyvariant specialization* [8], in which multiple specialized versions, or *variants*, of a program point may be constructed: program points are specialized with respect to the known values in their argument specifications, and cached specializations are re-used when all of the known values in their specifications match exactly.

Some implementations of this approach, such as [22, 40, 46, 16] increase the number and specificity of specializations by building them based on known types of otherwise unknown values. Others, such as [35], decrease the number of specializations without loss of accuracy via a more sophisticated caching mechanism.

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

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

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

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

[3] Another approach to dealing with representational issues such as code duplication and specialization-time structures in residual code is to delay some reduce/residualize decisions until after specialization is complete; see [44, 45] for details.

[4] In most specializers, these points are user function definitions, although some specializers, like Similix [6], have a prepass that adds additional function definitions to the program to enable specialization of program points that were not originally user functions.

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

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

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

2 Methods for Polyvariant Specialization

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

2.1 Offline Polyvariant Specialization

2.1.1 Description

Offline specializers are distinguished by their unwillingness to make reduce/residualize choices at specialization time. Instead, they make these choices prior to specialization, usually without full knowledge of the argument specification on which the specializer will be invoked. Along with the program and the argument specification, the specializer is given a set of directions (usually in the form of annotations on the program) which completely determine all the reduce/residualize choices it will make. Thus, unlike the naive strategy (which performs all possible reductions), an offline specializer will not examine the values of the subforms of an expression when deciding

[5] If we were to rewrite the program into a first-order form which passed environments explicitly and performed an explict `case` dispatch among the various function bodies reaching each call site, we could indeed build specializations on a per-call-site basis. We view this transformation as overly low-level because it forces a user-level representation of a virtual machine object, the environment. Such transformations may be counterproductive because they limit the Scheme compiler's ability to choose efficient machine-level representations. Once first-class environments become part of the Scheme language, program specializers will have more options when specializing programs such as the one above. For a comparison of several higher-order specialization techniques, see Section 4 of [38].

whether to reduce it. When the pre-ordained choice is "reduce," then it will, of course, use the values of subforms in performing the reduction, but it will never use such specialization-time information to dynamically choose whether to perform the reduction at all.

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

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

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

Under such an annotation scheme, not all annotations of a program are considered well-formed. The specializer must be able to perform the reductions specified by the annotations. It should never be the case that an expression which might receive an unknown argument in a strict position is marked "reduce," nor the case that a formal parameter which could be bound to an unknown value is marked "reduce." For instance, if the expression `(null? l)` in the example were marked "reduce," the specializer would have to decide whether the unknown value bound to `l` is the empty list, which isn't possible. Similarly, if the formal `ans` were marked "reduce," the specializer would needlessly build separate specializations for the calls `(length foo 0)` and `(length foo 100)` with `foo` unknown. Well-formedness criteria for annotations have been developed as part of the work on "congruence" described in in [26, 30].

Usually, these annotations are placed semi-automatically.[6] A prepass called Binding Time Analysis [27] (BTA) computes, for every expression, a conservative approximation of its specialization-time value, determining whether the expression is "static" (its value is guaranteed to be known at specialization time) or "dynamic" (is value might be unknown at specialization time).[7] These approximations are used to compute the "reduce" and "residualize" annotations:

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

[7] More sophisticated binding time analyses compute more detailed approximations, such as "the value will either be completely known (static) or will be a list of pairs, each of which has known (static) `car` and possibly unknown (dynamic) `cdr`." For details of such analyses, see [32, 11, 12, 30].

dynamic identifiers must be residualized, as must all special forms and primitive procedure calls with dynamic arguments in strict positions. Everything else may be marked "reduce." As we saw before, however, to provide folding and termination, certain calls (and, possibly, certain arguments to those calls) must be marked "residualize." Such additional annotation is usually performed via a variety of means, including manually (as in the original Mix [27] and the "generalization" operator of Similix [6]), automatically (the "dynamic conditionals" of Similix and the "finiteness analysis" of [23]), or via a meta-language (the "filters" of Schism [12]). The results of binding time analysis are also used by the cache lookup code in the specializer, which uses the binding time descriptions of the actual parameters to decide what to use as the key (the usual practice is to use the "static" parts of the parameters, instead of annotating the formals as in our example above).

2.1.2 Motivation

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

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

Finally, since an offline specializer's reduce/residualize behavior is completely determined by its program argument, but is valid for different specification arguments, it is possible to build a version of the specializer with the reduce/residualization behavior "built in," eliminating the need for interpreting the annotations. This has been accomplished to varying degrees by (1) moving more functionality into the annotations [14], (2) handwriting a compiler generator [25], and (3) self-applying the partial evaluator [19, 27]. Offline specializers are particularly amenable to self-application due to their small size, lack of an encoding problem, and statically determined reduce/residualize behavior. Making the reduce/residualize behavior depends only on the the program argument, which is known at self-application time, avoids any danger of losing that information at specialization time and turning a static choice into a dynamic one [7].

However, this efficiency does have a cost in accuracy (*c.f.* Section 3).

2.2 Online Polyvariant Specialization

2.2.1 Description

Online specializers are distinguished by their willingness to make reduce/residualize choices at specialization time. They do this much like one might imagine a "naive" program specializer operates: the specializer represents unknown values explicitly, so that it can tell whether or not `if`, application, and strict primitive expressions are reducible merely by examining their

[8] Some offline systems, particularly those handling higher-order functions or strongly typed languages, still require their own representations; see [31] and the description of *pe-closures* in [5].

argument values. In addition, such a specializer needs some mechanism to force folding and the creation of sufficiently general instances.

The explicit representation of unknown values requires, at minimum, adding a distinguished "unknown" to the type lattice of the language being partially evaluated. Usually, online specializers go further, extending the type system to allow unknown values to be placed in data structures and closures, and to provide unknown values with known attributes such as types or arithmetic signs [45, 16].

Online specializers use a variety of mechanisms to cause the building of specializations: some are based on explicit reduce/residualize annotations [20], others on a metalanguage [10], and still others on various forms of recursion detection [45, 43]. The meta-language approach allows the user to define a "fold here?" predicate for each function, which is passed the function's arguments and decides whether or not to unfold it. The recursion detection mechanisms compute call histories (essentially a representation of the pending calls in the interpreter) and fold whenever a recursive call can't be proven to be well-founded. Systems with a fixed "fold here" annotation have folding behavior identical to that of offline specializers, which must mark certain calls as "residualize"; systems with more dynamic mechanisms will fold less often, leaving fewer residual procedure calls. Premature folding results in extra procedure calls at runtime, but isn't a major problem in and of itself. What *is* important is how, after making the decision to fold, the specializer builds the specialization.

What distinguishes the specialization behavior of online specializers is how, after having decided to fold, they build suitably general instances of functions. Unlike offline specializers, which must pre-compute which reductions to perform while building the general function instance, online specializers achieve generality by modifying the argument values on which the specialization will be built, thus indirectly affecting which reductions will be performed during the construction of the general instance. This process is usually called *generalization* [43, 45], but has also been referred to as *generalized restart* [39]. After deciding to build a specialization, the specializer finds the set of call sites for which the specialization must be applicable, then computes the least general argument specification which includes the argument specifications of all relevant call sites, and builds the specification using the new, general, specification. Since argument specifications are just vectors of values, this generalization can be accomplished by computing least upper bounds in the domain of values. Generalization usually occurs in a *pairwise* manner; as new call sites are added to a specialization by the folding mechanism, the specification is recomputed, and the specialization is rebuilt. The important feature of online generalization mechanisms is that they only lose information (by computing more general specifications) when this is necessary; any information which is common across the call sites is retained. This allows, for instance, the maintenance of type information in loops with polymorphic parameters, and the building of minimally general specialized versions of higher-order procedures with multiple call sites.

2.2.2 Aside: online specializers that really aren't

Note that one can build an "online" specializer that makes the same reduce/residualize choices as an offline specializer would by simply running a binding time analysis in parallel with the specializer. This approach would construct the "directions" for reduce/residualize at specialization time, then obey them. One such a system, described in [20], performs BTA (called "configuration analysis" in [20]) on-the-fly before specializing each function. This analysis could have been performed statically by a polyvariant BTA; performing it dynamically achieves similar polyvariance with a simple analysis.

We would expect that such systems would be efficiently self-applicable, since, just as in the offline case, all reduce/residualize decisions are made by a process that refers only to the program text and statically available information (binding times), so there is no danger of losing this information at self-application time. Unfortunately, such methods have the same conservative behavior as purely offline methods, because they only use the "static" and "dynamic" approximations to known and unknown values when making reduce-residualize choices. Our further discussions of the online method will explicitly assume that such a "vacuously online" strategy is not in use.

2.2.3 Motivation

The motivation behind the online specialization method is that delaying the reduce/residualize choice until specialization time means that more information will be available when the choice is made, and thus the choice can be made more accurately. Efficiency considerations are considered secondary to the goal of performing as many reductions as possible at specialization time. Unlike the offline strategy, which performs generalization ahead of time, and can always build a specialization in a single pass, online specializers are willing to recompute specializations, possibly multiple times (as in [38, 37]), in order to build better residual programs.

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

The next section of this paper describes areas where online specialization either provides better accuracy than offline specialization, or provides similar accuracy with less need for "pre-transformation" of the input program.

3 Opportunities for Online

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

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

3.1 Representing Multiple Contexts with one Annotation

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

This behavior also occurs at specialization time, but with another twist: unlike evaluation, where there is always enough information to reduce any redex, the set of reductions performed may also vary based on whether sufficient information is available to reduce them. Unfortunately, in the offline paradigm, a particular source expression is annotated as either "always reduce" or "never reduce" (residualize). The inaccuracy here is obvious, since all specialization-time instances of an expression must be treated in the same manner regardless of the circumstances, any expression which might be non-reducible in some context must be left residual in all contexts. At runtime, this causes a penalty in the form of extra reductions that could be been performed at specialization time. Worse yet, failing to reduce an expression doesn't affect that expression alone: its return value will be lost, and any other expression that requires that value in order to be reduced will become irreducible as well.

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

Another approach to the problem doesn't attempt to perform the reductions at specialization time, but instead makes local reductions (reducing `if`s with known tests and beta-reducing applications with known heads which aren't part of loops, etc.) in a postprocessing phase. This recovers some of the lost efficiency by performing reductions before runtime, but doesn't regain all of it. In most cases, the postpass is not as powerful as a specializer; in particular, it lacks the ability to build new specializations based on information made available by the reductions it performs. To perform all possible reductions, one either requires a postpass with the full power of a specializer[9], or one has to iterate specialization. Such iteration has been suggested, but is not commonly performed because it would require running both the specializer and the annotation phase multiple times. Not only would this negate much of the efficiency advantage of offline specialization, but it would make systems with manual or human-assisted annotations difficult to use, due to the need to annotate the intermediate machine-generated programs.

In this section, we describe three situations in which the necessity to represent multiple contexts with a single annotation leads to less efficient residual programs, and describe how the online approach avoids these inaccuracies. The first such situation involves computing the reduce/residualize annotation for expressions that use the return value of a conditional expression which is reducible at specialization time. The second situation occurs in interpreters, which often contain expressions whose reducibility depends on specialization-time data, while

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

the third situation involves computing annotations for expressions which access data structures whose size is unknown until specialization time.

3.1.1 Return Values from Specialization-time Choices

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

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

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

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

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

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

If `(a = 0)` is the usual case, the residual program will perform many `+` reductions which could have been performed at specialization time. This can get worse: any expression surrounding the `+` would also have to be left residual; this sort of "chaining" can lead to large numbers of unnecessary residualizations (consider the case where `+` is replaced by "raise to the 435th power").

An offline specializer's performance can be improved by transforming the input program; such transformations are often called "binding time improvements." One method, due to Mogensen [32], distributes outer computations across inner ones. Such duplication of source code allows each context to be considered independently. In this example, the program would become

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

which would be annotated as

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

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

Another method, due to Consel and Danvy [15], performs CPS conversion on the program, yielding

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

which duplicates context by creating two entry points to the continuation instead of just one. The specializer can treat these entry points separately either by unfolding both invocations of `k`, or by building a different specialization at each call site. Unfortunately, CPS conversion introduces higher-order constructs; since all current BTA algorithms for untyped higher-order programs are monovariant, the expression `(+ a temp)` would not be duplicated, and would be marked residual, giving no improvement.[10] Consel and Danvy address this problem with BTA by duplicating the continuation at CPS conversion time, allowing the copies to be annotated separately.

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

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

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

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

```
(* a b)
```

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

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

3.1.2 Data-Dependent Contexts in Interpreters

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

This effectively prohibits the specializer from taking advantage of structure in the program being interpreted, yielding a "non-optimizing" compilation process. Consider a fragment from an interpreter for a trivial higher-order language with unary functions and binary primitive operators, as shown in Figure 1.

[10] This is only a problem with current BTA, not a problem with static annotation or the CPS strategy in general.

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

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

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

Figure 1: A sample interpreter for higher-order programs

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

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

The binding time improvement techniques used above don't work here, as they rely on duplicating expressions to duplicate context. In this case, we need to duplicate the appropriate arm of the `case` for every expression in the program being interpreted; this isn't possible at annotation time because the program isn't available yet. Bondorf [4] solves the problem by cleverly rewriting the interpreter to represent interpreter closures as Scheme closures (*i.e.,* replace (`make-closure formal body env`) with (`lambda (arg) (eval body (extend-env env formal arg))`) and rewrite `apply` appropriately), effectively moving the recursive call to `eval` from `apply` into `make-closure`. This trick keeps the syntactic parameter `exp` from becoming dynamic, but doesn't allow the reduction of `if`s (or, for that matter, function applications) at specialization time. Accomplishing such reductions requires iterating specialization, as described above.

Once again, an online specializer (in principle[11]) has no problems with this example because it makes its reduce/residualize decisions at specialization time, when the necessary information is available.

3.1.3 Aggregates

Static annotations complicate reasoning about aggregates, such as lists and sets, whose sizes and element values are not known until specialization time. At specialization time, the elements of a list may be completely independent of one another; some may be known and others unknown. Given a loop (say, the function `map`) traversing a list and performing some operations on the elements, we would like to perform the operations on the known elements and leave them residual on the unknown elements; furthermore, in the case of a list with an unknown tail, we would like to unfold the loop until that tail is reached, and then build a residual loop.

These behaviors cannot always be achieved under a static annotation strategy. The decision of whether to reduce or residualize the elementwise operations must be encoded on the source program, which is only possible if we know the length of the list and the known/unknown status of each element of the list at annotation time; otherwise, we must annotate all of the elementwise

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

operations as "residualize," losing all knowledge of the known elements. This is similar to the interpreter case in Section 3.1.2, in which we were unable to distinguish the environment binding of `actual`, which was unknown, from other bindings which might be known.

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

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

In some cases, this behavior might be adequate if the annotation strategy could handle lists whose construction is completely determined by the source program, even if it couldn't handle lists built under static control at specialization time. An example of such a structure might be one which doesn't depend on any data supplied at specialization time, such as an initial environment frame or other initial state built by an interpreter or simulator. Current methods for computing annotations don't take advantage of concrete information (constants, etc) in the program being annotated; these get abstracted just like the inputs do. This is one reason why specializing a program on completely unknown inputs (thus performing reductions based on concrete values in the program), then recomputing the annotations, can be helpful.[12]

3.2 Inability to take advantage of Commonality

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

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

Often, the specializer is forced to build a single approximation which represents several possible runtime executions. Doing this may require merging existing approximations (*i.e.,* computing a single approximation which denotes at least the union of the denotations of the

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

approximations being merged; typically, the approximations are taken from a type lattice, and merging two approximations is equivalent to computing their least upper bound.). For instance, the value returned from a residual `if` expression could be the value of either arm; similarly, the value of a formal parameter of a specialization could be the corresponding actual parameter from any of the specialization's call sites.

Because generality forces more reductions to occur at runtime, we would like to build the least general residual expression which is still sufficiently general. One obvious method is to discard only that information which differs between contexts, and to use the (remaining) common information in performing specialization. This is the motivation behind the generalization strategy often used in online specializers. Unfortunately, finding this sort of commonality requires testing whether specialization-time values (and their subparts) are equal. Such testing isn't explicitly outlawed by the offline method—even offline polyvariant specializers compare static argument values in the cache in order to achieve re-use of specializations and thus folding. However, using the results of such an equality test for the purpose of making a reduce/residualize decision *is* prohibited. That is, an offline specializer may not make use of an value obtained by merging others unless it can prove, at annotation time, that all of the values being merged will be equal at specialization time. If this were not the case, the specializer would have to examine the merged value (to determine if it was known or unknown) in order to decide whether to perform reductions depending on it. In many cases, it is not possible to perform such a proof at annotation time; even in cases in which it might be possible to perform such a proof, all current annotation methods fail to do so, and thus all return values and parameters to residual higher-order procedures are always declared to be "dynamic," forcing residualization of all references to them.

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

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

3.2.1 Return values of `if` expressions

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

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

where `a` is unknown, and both `b` and `c` are known. If `b` and `c` have the same value, then we can reduce the `if` expression to that value, and use it in reducing the enclosing expression; otherwise, we must leave the `if` residual, and can only use information common to the values of both b and c in reducing the enclosing expression. Of course, it's unlikely that the variables b and c will have the same value, but it is often the case that their values will share some common properties. For instance, programmers often construct "ad hoc" types by constructing tagged pairs; it is worthwhile to take advantage of the equality of tags in these situations. For example, if b=(foo-type . <unknown>) and c=(foo-type . <unknown>), returning (foo-type . <unknown>) instead of <unknown> will allow the specializer to reduce type tests of the form (eq? (car (if (= a 0) b c)) 'foo-type).[13] Similarly, in specializers which compute types or other static properties of unknown values [46, 16], one would like to retain type information common to both branches of a residual if expression. For instance, it might be useful to know that both arms of an if are integers when reducing an enclosing application of +.

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

The context duplication methods (code copying and CPS conversion) described in Section 3.1.1 work in this case, because they distribute consumers of conditionally known values across the conditional. For example,

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

is transformed to either

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

or

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

However, unlike the previous case in which the if was reduced to one of its arms at specialization time, in this example, both arms will appear in the residual program, so context duplication will result in duplication of code. In cases where the known portions of the two

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

arms differ, this is indeed desirable, as it will allow all of the information in both arms (rather than just the information common to both) to be used in reducing the application of `foo`, but in cases where the known portions are identical, this duplication will serve no useful purpose. This naturally suggests duplicating the context only when the generalization loses information; such an optimization is possible under an online strategy, but not under an offline one due to the necessity of making reduce/residualize decisions based on a specialization-time comparison.[14] As was the case before (*c.f.* Section 3.1.1), loops can make it impossible to perform such context duplication statically; we will see an example of this in the next section.

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

3.3 Return values of specializations

Another example of context sharing occurs at residual call sites: the expression(s) surrounding the call must be able to handle any values returned by the call. For each residual call to a specialization, the specializer must decide which of the surrounding expressions are reducible, and which must be residualized. One simple tactic assumes that all references to the return value of a residual call must themselves be left residual; this is obviously safe, but misses reductions not only at calls to the specialization, but, in the case of specializations of recursive functions, inside the specialization itself. Better approximations can be returned if generalization is performed, but this requires performing comparisons which are forbidden under the offline scheme.

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

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

specialized on `n` and `ans` specified as unknown integers.[15] We would like to retain the information that `n` and `ans` remain integers on the recursive call, and that the return value is an integer. Retaining the type of `n` and `ans` is simple: an annotation mechanism can prove that their types are static; thus, the type information will be retained when the specialization is built, potentially allowing the use of `=`, `-`, and `*` operators with fewer tag checks. No annotation-time knowledge of type equality is necessary, because the specializer will simply build a new variant if the types of `n` or `ans` change on the recursive call.

[14] To date, no online specializer implements such an "on-the-fly" duplication strategy; the point is that such an optimization is possible only in an online framework. Offline specializers can achieve this behavior via postprocessing: Similix-2[4] handles this case by forcing all `lambda`-bodies to be specialization points (in this case, building either one or two specializations of `k`, depending on whether `c=d`), then unfolding some of those specializations in a postpass (if `c=d`, the specialization will not be unfolded, keeping sharing; otherwise, both specializations will be unfolded).

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

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

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

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

into either

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

or

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

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

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

Another way of viewing this is to note that both of these techniques work by converting an upward-passed value into a downward-passed value to avoid losing information at a point where approximations must be merged (the `if` statement). Information about downward-passed values is never lost, because specializations are shared only when their static arguments match exactly; different values result in a new specialization instead of a more general one. Unfortunately, for truly recursive programs expressed in CPS style, termination often requires that multiple call sites with different static arguments share the same specialization, necessitating online comparisons and reduce/residualize decision making to retain information. This problem is the subject of Section 3.3.1.

An online specializer can handle these return values quite naturally; instead of transforming the program to avoid the problem (and thus introducing other problems), it is willing to build an initial approximation to the return value, then weaken it later as necessary. For details of how this is accomplished, see [46, 38].

To show that this behavior occurs in contexts other than scalar type inference, our second example comes from the domain of interpreters. The code fragments in Figure 2, implement part of an interpreter for a "toy" imperative language. This interpreter represents the store as a list of pairs, each of whose `car` is an identifier and whose `cdr` is its value.

We can "compile" a statement by specializing the function `eval-stmt` on a known statement and an initial store containing known identifiers and unknown values. We would like the specializer to determine that the "shape" of the store parameter never changed; that is, it will always contain the name known identifiers, in the same order, as the store on which `eval-stmt` was initially specialized. Retaining this information allows variable lookups to be reduced to simple list accesses[16] without any need for comparing identifiers. There should be no residual loops which search for bindings in the store.

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

Just as in the case of the factorial example, an online specializer gives the desired result because it can compare stores, discover that their shapes are the same, and return a generalized store upward. Offline specializers, which cannot perform such comparisons, cannot handle the interpreter directly; this is a recognized open problem in offline partial evaluation [30]. There are several transformational approaches to solving this problem. One approach separates the store into two lists: one which holds the names, and need only be passed downward, and one which holds the values, and is passed both upward and downward. Now even if upward-passed values are approximated by "unknown," the list of names will not be lost. Of course, any information about the values in the store (such as their types) will be lost. Such rewriting is often performed by hand, but automated versions have been constructed [32, 18]; these have the disadvantage of being able to preserve only values which can be proven, at annotation time, to be known at specialization time; any data-dependent information will be lost.

Another approach rewrites the program so that it only passes the store downwards, never upwards. This approach is taken by Mogensen [32] and by Launchbury [30], whose interpreters

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

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

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

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

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

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

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

Figure 2: Fragments from direct-style MP interpreter

pass an extra argument containing the "rest" of the MP+ statements to be executed. Instead of returning a store when the loop is complete, `eval-while` exits by calling `eval-stmt` on the "next statement" and the store. This approach involves considerable hand-rewriting, and may not generalize well to other programs.

The third common work-around, which works well for interpreters for languages with iteration but without recursion, is to perform the CPS transformation on the program, as is done by Consel and Danvy [15]. After this transformation is applied, the store is passed as a parameter to continuations instead of being returned upward, eliminating the need for accurate approximations to returned values.

Unfortunately, the CPS transformation fails to preserve the shape of the store when we specialize the interpreter of Figure 2 on a program containing recursive procedure calls. In this case, the residual interpreter will contain continuations (residual `lambda` expressions) which must be applicable from multiple call sites with different known argument values; retaining the information common to their call sites cannot be accomplished by binding time improvement techniques, because the equality of the known argument values, which is needed at annotation time, cannot be tested until specialization time. We will discuss this problem further in Section 3.3.1. Online methods handle programs with procedure calls without difficulty, because they can preserve the shape of the store directly by computing generalized return values as necessary.

3.3.1 Parameters of specializations

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

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

As we saw in Section 1.2, a real specializer may have to build a single specialization which is applicable at multiple call sites which have different argument vectors. First, the specializer will only terminate on a loop if the initial and recursive entry points of the loop call the same specialization. Second, a specialization of a higher-order function must be applicable at all call sites which might be reached by that specialization at runtime.

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

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

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

Offline specializers accomplish this by deciding, at annotation time, which portions of a function's arguments to use when building specializations and comparing them in the cache; to ensure termination, the annotation phase must assure that the portions which are used will assume a finite number of values at specialization time. In this case, the specializer can use the type of `ans` when building the specialization, but not its value. The main problems are that proving finiteness may be difficult, and that the finiteness of an argument might depend on specialization-time data; in both cases, failure to make use of argument values can lead to overly general specializations. This is less problematic than one might expect, because many offline specializers rely on the user to provide the generalization annotations. Users can solve the first problem by performing "proofs" themselves, and the second by being willing to make use of knowledge about specialization-time values, and making annotations that are valid only for these values. Often, this is discovered empirically; if the the specializer fails to terminate in a reasonable amount of time, the user goes back and adds some generalization annotations.

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

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

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

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

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

Unlike the case of upwardly returned values, in which the need to compute a single, general return value can be avoided by duplicating context, no amount of context duplication is sufficient to eliminate the need to compute a single, general specialization of each continuation in this case. A typical offline annotation phase will deduce that both the type and value of the parameters `ans` and `temp` will be known at specialization time, but will be unable to deduce that the values will vary between call sites while the types remain the same. Unless the annotation phase can prove that the types will remain the same, an offline specializer will build overly general specializations which don't take advantage of the type information. In the case above, the proof is fairly simple; a sufficiently smart annotation pass could indeed perform the proof, and generate the desired specialization, although, to date, no such annotation pass has been implemented.[17]. Proofs can become more difficult; proving that all residual continuations in an interpreter are invoked on a store of identical shape requires several inductive steps.

In the worst case, a proof is impossible because the commonality between call sites is dependent on specialization-time data. The `fact` example in Section 3.3 is such an example; as another simple example, consider a version of the `length` function in which the base case is not specified until specialization time:

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

If we specialize the program with `base=0`, the specializations of the initial and recursive continuations can use the fact that their parameters are integers; if we specialize it with `base=1.5`, then the specializations may only use the fact the their parameters are numbers. Unfortunately for offline specializers, both values for `base` have identical abstractions: "dynamic" value but "static" type, which isn't enough information to perform the requisite equality comparison. This example is deliberately trivial; such data dependent commonality does arise in the context of polymorphic programs, programs with error values, and interpreters for such programs.

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

Consider the interpreter of Figure 2. Specializing the CPS-transformed version of this interpreter on a program with `while` loops but without procedure calls is not problematic; only the procedure `eval-while` will be residualized, and it is tail recursive, meaning that its continuation will be beta-reduced at specialization time. Any store accesses in that continuation will be able to take advantage of the shape of the store. Specializing the CPS-transformed interpreter on a program with recursive procedure calls is different. In this case, the procedure `eval-stmt` will be residualized, and its specialization will contain a non tail-recursive call to `eval-stmt`, due to the unfolding of `eval-begin`. The residual program might contain code like

[17] Such an annotation pass would have to make use of concrete values in the program, will all of the relevant complications

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

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

An online specializer can handle both the `length` and the interpreter examples correctly because it is willing to iteratively compute generalized values for the continuation parameters `temp` and `ans` at specialization time (an algorithm for doing this is given in [38]). In cases where an offline system (or its user) would have to construct a difficult proof, an online system can provide equivalent output with significantly less effort, and in cases where data dependence is present, it can achieve results which are not possible under the offline method.

4 Improving Program Specialization

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

4.1 Offline

Offline specialization and binding time analysis were first developed by the Mix [27] project at DIKU, as a means of achieving self application. Since then, the accuracy of offline specialization has steadily increased, due to the development of both more accurate binding time analyses and various program transformation strategies called "binding time improvements."

Mogensen [32] developed a binding time analysis that handled structured data; Consel [11] developed a polyvariant version. The binding time analyses for higher-order languages due to Bondorf [4] and Consel [12] are monovariant only. More recent work on *facet analysis* by Consel and Khoo [16] has further increased the accuracy of binding time analysis by allowing it to make use of known properties of unknown values. None of this work, however, solves the fundamental problems with annotations themselves: certain information is simply not available until specialization time, and cannot be made available at annotation time.

Binding time improvement via the replication of code is common practice in writing interpreters to be specialized via offline means (for some example interpreters written in this style, see [5]). Automating such replication was first suggested by Mogensen [32]. Using the CPS transformation to perform the same task was first suggested by Consel and Danvy [15] and has been used to improve binding times in several examples [13, 29]. Other manual transformations, such as eta-conversion, are also common; Holst and Hughes [24] suggest a possible means of automating such transformations.

4.2 Online

The early program specializers of Kahn [28] and Haraldsson [22] were, to some degree, online specializers. Turchin's supercompiler [42], which can be considered a superset of polyvariant specialization, performs online termination analysis, folding, and generalization. Recent work on online specializers for functional languages includes the MIT scientific computing project [3, 1, 2], the Stanford FUSE project [45], and others [21, 40, 20]. Online methods are also popular in the logic programming community [39, 41].

Recent accuracy gains made in online specialization have been due mainly to specializers' ability to represent more detailed information about runtime values (the *partials* of [40], the *placeholders* of [3], the *symbolics* of [45, 46] and the *facets* of [16]), their willingness to use this information in a first-class manner (early specializers such as [22] severely limited the use of type information in data structures), and the use of generalization and fixpointing techniques to retain more information across residual calls [46, 38, 39].

Other useful advances in online specialization have dealt with representational issues, such as preventing code duplication without compromising accuracy by delaying some reduce/residualize decisions until after specialization [44, 45], and limiting the construction of redundant specializations through the use of type information [35, 36]. The former optimization is orthogonal to the choice of offline/online method, but the latter explicitly requires the ability to make reduce/residualize decisions at specialization time, and so is applicable only in online specializers. Termination mechanisms are also an active area of research; for some recent strategies, see [46, 39, 17].

Conclusion

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

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

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

In all of our examples, it has been the case that online specializers provide equal or better accuracy with less need for "pre-transformation" of the input program. We believe this represents useful progress toward an eventual goal of building good specializations of arbitrary user programs with minimal user intervention. This increased accuracy comes at a cost in performance; it is our hope that some of the techniques used to provide efficient offline specialization will be transferable to the online world. We also expect that many of the techniques used by binding time analyses to reason about specialization-time data structures at annotation time may be useful for reasoning about runtime data structures at specialization time, resulting in even more accurate specializations. Conversely, the offline methodology might be improved by using online mechanisms to utilize concrete values in the program at annotation time, or to make some termination-related decisions at specialization time. The question for the future isn't "which is better: online or offline?" but rather "how can we construct a system with the advantages of both worlds?"

Acknowledgements

The first author would like to thank Jim des Rivieres, Gregor Kiczales, John Lamping, Luis Rodriguez, and Carolyn Talcott for their comments on earlier versions of this paper.

References

[1] 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.

[2] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990.

[3] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[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. Consel. New insights into partial evaluation: the SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming*, pages 236–246, Nancy, France, 1988. Springer-Verlag, LNCS 300.

[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. From interpreting to compiling binding times. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, pages 88–105. Springer-Verlag, LNCS 432, 1990.

[15] 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.

[16] 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.

[17] R. Conybeare and D. Weise. Improved specialization and consistent termination via observing computation, or derivations are more useful than values. (in preparation).

[18] 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.

[19] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[20] R. Glück. Towards multiple self-application. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 309–320. ACM, 1991.

[21] 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.

[22] 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.

[23] C. K. Holst. Finiteness analysis. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*, pages 473–495. ACM, Springer-Verlag, 1991.

[24] 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.

[25] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[26] 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.

[27] 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.

[28] K. M. Kahn. A partial evaluator of Lisp programs written in Prolog. In M. V. Caneghem, editor, *First International Logic Programming Conference*, pages 19–25, Marseille, France, 1982.

[29] 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.

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

[31] 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.

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

[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. Avoiding redundant specialization during partial evaluation. Technical Report CSL-TR-92-518, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[37] E. Ruf and D. Weise. Improving the accuracy of higher-order specialization using control flow analysis. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, San Francisco, CA, 1992. (to appear).

[38] 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.

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

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

[41] D. Smith and T. Hickey. Partial evaluation of a clp language. In *Logic Programming: Proceedings of the 1990 North American Conference*, pages 119–138. MIT Press, 1990.

[42] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[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.