# Avoiding Redundant Specialization during Partial Evaluation*

**Erik Ruf and Daniel Weise**

**Technical Report: CSL-TR-92-518**
(also FUSE Memo 92-9)

April, 1992

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

## Abstract

Existing partial evaluators use a strategy called *polyvariant specialization*, which involves specializing program points on the known portions of their arguments, and re-using such specializations only when these known portions match exactly. We show that this re-use criterion is overly restrictive, and misses opportunities for sharing in residual programs, thus producing large residual programs containing redundant specializations. We develop a criterion for re-use based on computing the domains of specializations, describe an approximate implementation of this criterion based on types, and show its implementation in our partial evaluation system FUSE. In addition, we describe several extensions to our mechanism to make it compatible with more powerful specialization strategies and to increase its efficiency. After evaluating our algorithm's usefulness, we relate it to existing work in partial evaluation and machine learning.

**Key Words and Phrases:** Partial Evaluation, Program Specialization, Polyvariant Specialization, Re-use Analysis, Explanation-Based Learning.

*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, resulting in a specialized program that runs faster than the original program did.

Program specializers operate by symbolically reducing the program on the specification of its inputs. Computations that can be performed, given the information available, are performed; otherwise, *residual* code is generated, delaying the computation until the specialized program is run. Not all performable computations are performed: to guarantee termination of the specializer, and to provide sharing in specialized programs, the specializer performs *folding*[8] operations. Folding is done by recursively specializing certain distinguished[1] parts of the program (*specialization points*), and re-using these specialized subprograms (*specializations*) where appropriate. Most specializers use a strategy called *polyvariant specialization*[7], in which program points are specialized with respect to the known values in their argument specifications, and specializations are re-used when all of the known values in their specifications match exactly. Some variants of this method, such as [23, 41, 49] go further, allowing the building of specializations based on known *types* of otherwise unknown values.

In this paper, we examine this practice of re-using specializations of program points based on matching argument specifications. We contend that the argument values on which a program point is specialized often contain more information than is actually utilized in the computation of the specialization. That is, *different* argument values can produce the *same* specialized procedure. This behavior hurts performance both at program specialization time (by forcing the partial evaluator to compute specializations needlessly), and at runtime (by building unnecessarily large residual programs, which cause degradations in cache and virtual memory performance).

Before we consider examples, a few explanations are necessary. The examples in this paper involve specializing programs written in a functional subset of Scheme[34]; thus, both programs and specialization points will be user function definitions. This is true of most specializers for functional languages, although some, 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. We will treat the terms "program specializer" and "specializer" as synonyms. To denote the amount of information the specializer is given about a value, we use the terms "known" and "unknown," rather than "static" and "dynamic," because these latter terms may connote approximations; in systems using binding time analysis[26], "dynamic" means "possibly unknown" rather than "unknown." Finally, to avoid confusion, all of the specializations in the examples take the same number of arguments as the function definitions that were specialized; of course, a real specializer would perform arity reduction to remove dead formals, and would hoist invariant formals to an enclosing lexical scope.

Different argument values can produce the same specialization in several ways. Consider specializing[2]

[1]Many program specializers, such as Mix[26] and its descendants, make the choice of these points statically, before specialization takes place, but it is also possible to make this decision dynamically during specialization, as is done in FUSE[47] and Turchin's supercompiler[45].

[2]We realize that, under most specializers, this function (and many of our other examples) would never be spe-

```
(define (test1 x y z)
  (if x
      (+ 1 z)
      y))
```

on `x=#t`, `y=2`, and `z` unknown; the residual code would be

```
(define (res-test1 z)
  (+ 1 z))
```

This specialization will only be re-used at call sites where `x=#t`, `y=2`, and `z` is unknown. If, in the same program, `test1` is also called with different values for `y`, the specialization `res-test1` will not be re-used; instead, a new, *identical* specialization will be built for each value of `y`, and all of these specializations will appear in the residual program.[3] Ideally, the specializer should deduce that all specializations with `x=#t` and `z` unknown are equivalent; the value of `y` is dead, and thus should not be factor in the decision whether to re-use the specialization `res-test1`.

This behavior occurs in cases other than the "dead formal" case above. A specialization can depend on only part of a structured parameter, such as pair, vector, or closure. Specializing

```
(define (test2 x a)
  (+ (car x) a))
```

where `x` is a pair with a `car` of `1`, and `a` is unknown, yields a specialization that is independent of the value of `cdr x`. A naive specializer would build different specializations for `x=(1 . 2)` and `x=(1 . 3)`. Similarly, a specialization can depend only on the type of an argument, rather than its value; if `a` and `b` are unknown, the specialization of

```
(define (test3 x a b)
  (if (number? x) a b))
```

depends only on whether `(number? x)` is true, not on any specific value of `x`. In other cases, type information about a parameter is not used in building a specialization; specializing

```
(define (test4 x)
  (if (null? x)
      0
      (1+ (length (cdr x)))))
```

cialized, but only unfolded. However, by introducing recursion, we can trivially convert all of our examples to ones which must be specialized in order to guarantee termination of the program specializer. For the case of the `test1` function in this example, consider instead specializing

```
(define (must-specialize a x y z)
  (if (null? a)
      '()
      (cons (if x (+ 1 z) y)
            (must-specialize (cdr a) x y z))))
```

on `x=#t`, `y=2`, and `a` and `z` unknown.

[3]In this particular case, a postprocessor could presumably eliminate the extra definitions, but it is possible to construct examples where a reasonable postprocessor could not. Also, even if a postprocessor could improve the residual code, the time wasted building redundant specializations could not be reclaimed.

on a list `x`, of unknown length, yields the same result regardless of any knowledge about the type of the elements of the list. Finally, parameters can interact; specializing

```
(define (test5 x y a)
  (* (+ x y) a)
```

on `x=1`, `y=2`, and `a` unknown yields

```
(define (res-test5 x y a)
  (* 3 a))
```

which is valid for any values of `x` and `y` whose sum is `3`, not just for the specific case of `x=1` and `y=2`.

Thus, different sets of argument values can produce the same specialization; conversely, a specialization can often be safely applied to sets of argument values different from those used to build the specialization. If we define the *domain* of a specialization as the set of argument values for which it returns the same result as the original function, we can state the problem more clearly: *a specialization of a function often has a domain which is larger than the set of argument values denoted by the argument specification used to build the specialization.* We advocate reasoning about the domains of specializations in order to re-use specializations more effectively. Of course, specifying some domains, such as that of `res-test5`, would require a constraint solver for real arithmetic, which is far beyond the scope of the approximate solution we offer.

This paper has five sections. The first develops an informal theory of specialization, and provides a criterion for choosing when to re-use an existing specialization. Section 2 describes our partial evaluator, FUSE, its type system, and how an approximate version of the re-use criterion is implemented using the type system. This is followed by a description of some extensions to the re-use mechanism (Section 3), and discussion of where, during the specialization of everyday programs, our mechanism is useful (Section 4). We conclude with a discussion of related work (Section 5).

# 1 A Re-Use Criterion for Specializations

This section describes a criterion for optimal re-use of specializations, and comments on why this criterion is not directly implementable.

## 1.1 Formalisms

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

We define a specializer as follows:

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

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

This definition 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.

In order to build a specialization of a particular function, it is often the case that the specializer will build other specializations (possibly of the same function) as well. To avoid duplication of work (and to guarantee termination of the specializer), most specializers use some form of caching. The class of *polyvariant* specializers re-uses specializations with respect to identical specifications. When the specializer specializes a function $f$ on some argument specification $s$, it makes an association between $f$, $s$, and the specialized (residual) function $r$, so that subsequent attempts to specialize $f$ on $s$ (or any specification $x$ satisfying $C(x) = C(s)$) can simply return $r$ without further computation. This form of re-use is obviously safe, since if the specialization $r$ is valid for all values denoted by $s$, it is surely valid with respect to some other specification that denotes an identical set of values. We will refer to $s$ as the *index* of the specialization $r$.

Sometimes, as in the examples above, when a function is specialized, not all of the information in the argument specification is used in computing the specialization. Re-using specializations only when the argument specifications are identical misses opportunities to re-use specializations because the argument specification does not always accurately reflect the set of values over which the specialization is valid. That set is defined as follows:

**Definition 2** *(Domain of Specialization) If $f$ is a function definition, and $r$ is the result of specializing it on some argument specification, then the* domain of specialization *of $r$ is a set of values $DOS(r){\in}\mathcal{PS}(V)$ where*

$$DOS(r) = \{v{\in}V \mid E(f,v) \neq \perp_V \;\Rightarrow\; E(f,v) = E(r,v)\}.$$

The domain of specialization is the set of argument values for which the specialization and the original function definition compute the same result, or under which the original function fails to terminate. As before, we allow the specialization to return any value on argument values which cause the original function definition to fail to terminate. The DOS is useful as a safety criterion:

**Definition 3** *(Safe Re-Use) When a specializer is faced with the task of specializing a function $f$ on a specification $s$, it can* safely re-use *another specialization $r$ of $f$ whenever $C(s) \subseteq DOS(r)$.*

This is a correctness criterion only, and does not guarantee optimality. Consider specializing `(lambda (x y) (+ x y))` on unknown `x` and `y`, producing `(lambda (x y) (+ x y))`. The correctness criterion indicates that re-using this specialization for the case `x=1, y=2` is safe; however, it does not indicate that not re-using the specialization will allow us to build a "better" one, namely `(lambda (x y) 3)`. The domain of specialization doesn't indicate whether the specializer could generate a "better" specialization for a smaller domain.

Before proceeding, we must codify this notion of "better." Intuitively, the goal of a specializer is to perform reductions at specialization time, so that these reductions will not be performed at runtime. Because these reductions make use of information available at specialization time, that information must also be true of the values passed in at runtime. Thus, each time the specializer makes use of specialization-time information to perform a reduction (and thus build a more efficient specialization), it reduces the domain of the specialization. Making use of this observation, we can define an optimality relation on specializations:

**Definition 4** *(Strictly More Specialized) A specialization $p$ of a function $f$ is* strictly more specialized *than another specialization $q$ of $f$ if and only if $DOS(p) \subset DOS(q)$.*

Thus, for `x=1`, `y=2`, the specialization `(lambda (x y) 3)` is strictly more specialized than the specialization `(lambda (x y) (+ x y))` because it has the domain of pairs of numbers whose sum is 3, rather than the domain of pairs of numbers.

Now that we have both a safety criterion and an optimality criterion, we can formulate a re-use criterion:

**Definition 5** *(Optimal Re-use) Given a function definition $f$ and argument specification $s$, a specializer may* optimally re-use *a specialization $r$ of $f$ if and only if it is safe (Definition 3) to do so, and $SP(f, s)$ is not strictly more specialized (*i.e. *not a better specialization) (Definition 4) than $r$. In other words, $r$ can be optimally re-used if and only if*

$$(C(s) \subseteq DOS(r)) \wedge \neg(DOS[SP(f, s)] \subset DOS(r)).$$

### 1.2 Practicalities

Given a re-use criterion for specializations, what can we do with it? As it stands, the definition has two major practical deficiencies:

1. The real specializer is a program, not a mathematical function. It cannot manipulate arbitrary, potentially infinite sets of values (such as the domain of specialization or concretizations of argument specifications) directly; instead, it must operate on finite, approximate representations of these sets. We will use elements of a type lattice as representations; this will allow us to recast all of the above definitions using domain operators instead of set operators. Although the accuracy of approximation will depend on our choice of type lattice; the correctness of our algorithms will not depend on any particular choice.

2. The optimality test requires that the specializer compute the specialization $SP(f, s)$ before deciding whether to re-use a pre-existing specialization $r$ of $f$. While this method produces the desired output, it does not satisfy our goal of saving work in the specializer itself, since the specializer would need to build a specialization in order to determine whether it is necessary to build that same specialization. Since, in some cases, it is possible to prove that $DOS[SP(f, s)] = DOS[r]$ from $r$ and $s$ alone, without computing $SP(f, s)$, our strategy will be to attempt this proof, and apply the optimal re-use criterion only when the proof fails.

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

Figure 1: Value domains for Scheme

## 2 Re-use of Specializations in FUSE

The remainder of this paper describes details of FUSE[47], an on-line program specializer for a side-effect-free subset of the programming language Scheme, with atoms, pairs, and higher-order procedures, but no vectors, input/output, first-class continuations, or `apply`. Given these restrictions on types, the syntax and semantics of the language processed by FUSE are very similar to those in [34] and will not be described here.

### 2.1 Specifying Values with Types

A specializer operates on programs and argument specifications. Because Scheme functions can take multiple arguments, we will use *value specifications* to specify the values of individual arguments; an argument specification is simply a tuple of value specifications. A value specification is a finite description of a possibly infinite collection of values that might appear in its place at runtime. Because such a description is finite, it is necessarily approximate; to ensure correctness, it must also be conservative. That is, any specification must denote at least those values which will be passed as arguments at runtime; often it will denote more values.

The definition of a value specification corresponds nicely with our notion of a *type*, since types are also used to describe sets of possible values. Thus, it seems intuitive to use types as value specifications. We immediately run into a problem, however—Scheme is an untyped language, and there are many possible type systems that we could impose upon it.

A specializer for a language can be considered as an interpreter for the same language that operates under a nonstandard semantics. Making this observation, we note that a typical Scheme evaluator manifestly types the values it operates on. The types it uses are exactly those which are injection tags in the value domain of Scheme, namely those shown above in Figure 1. These are the types used by the evaluator and the primitive functions; since the specializer is, in some sense, emulating the evaluator, we would like it to be able to describe the types used by the evaluator, so that it can make reductions based on type information.

FUSE's domain of value specifications is similar to the domain *Val* in Figure 1. The main difference is that, while an evaluator operates only on concrete values (such as `1`, and `(foo . bar)`), the specializer must also be able to describe sets of values. One way to do this is to add top elements to the the various subdomains in *Val* (*i.e.,* "lift" the CPOs), since each top element can be interpreted as describing all of the elements below it. Therefore, in addition to values, FUSE

```
;;; This figure is a transcript; lines beginning with "=>" were typed by the user
;;; Other lines were printed by the Scheme evaluator

=> (pp (sval-value-spec (scheme-value->sval 2)))
#(type #type-mk %number)
  (value 2)

=> (define s (sval-value-spec (scheme-value->sval `(,(a-value) . ,(a-number)))))

=> (pp s)
#(type #type-mk %pair)
  (value
  (#[SV 1055]# . #[SV 1056]#))

=> (pp (sval-value-spec (car (value-spec-value s))))
#(type #type-mk %top)
  (value ())

=> (pp (sval-value-spec (cdr (value-spec-value s))))
#(type #type-mk %top-number)
  (value ())
```

Figure 2: Examples of value specifications

value specifications may also contain representations of $\top_{Num}$, $\top_{Bool}$, $\top_{Sym}$, $\top_{Str}$, $\top_{Func}$, and $\top_{Val}$. There is no need to represent $\top_{Nil}$ or $\top_{Pair}$, since *Nil* has only one element, and because the "least known" pair can be represented as $< \top_{Val}, \top_{Val} >$. FUSE defines a partial order on value specifications in the usual way.

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

In actuality, the FUSE type system is slightly more complicated. In order to support fixpointing, a representation for $\perp_{Val}$ is available, as is a representation of $\top_{List}$, which denotes a pair or the empty list. In the future, we expect to add support for recursive datatypes, and limited support for disjoint union types.

To make these ideas concrete, we discuss FUSE's representations. FUSE operates by interpreting function definitions under a nonstandard semantics and nonstandard value domain. Instead of using ordinary Scheme values, FUSE uses *symbolic values*. A symbolic value has several attributes, one of which is a value specification, denoting the values that could possibly appear in place of the symbolic value at runtime. Several other attributes are used by the code generator, and will not be discussed further here (they will also be omitted from examples). The value specification consists of a type marker and an optional value attribute. The type markers `bottom`, `top`, `top-number`, `top-boolean`, `top-string`, `top-proc`, and `top-list` do not denote specific values, and thus do not take a value attribute, while the type markers `number`, `boolean`, `symbol`, `string`, `prim-proc`, `compound-proc`, and `pair` all take a value attribute, which is the actual number, boolean, symbol, string, primitive procedure, compound procedure, or pair that is denoted. In the case of pairs, the specification denotes all pairs whose `car` and `cdr` are denoted by the symbolic values in the `car` and `cdr` positions, respectively, of the value attribute. For examples of value specifications, see the transcript in Figure 2 or [49].

## 2.2 Computing Approximations of the DOS

Now that we have described FUSE's repertoire of argument specifications, we turn our attention to using them to compute the domain of specialization of a particular specialization. As mentioned before, the DOS is often infinite, and cannot be computed directly. Instead, we compute an argument specification which safely approximates the DOS, in the sense that its concretization is a subset of the DOS. We say an argument specification $x$ is *strictly more general* than another specification $y$ if $x \sqsupset y$. When $x$ is strictly more general than $y$, we also say that $x$ has *strictly less information* than $y$, or conversely, that $y$ has *strictly more information* than $x$. For a specialization $r$ of a function definition $f$, the approximation we are seeking is the most general argument specification $s$ such that $SP(f, s) = r$. We will call this specification the *Most General Index*, or MGI, of the specialization $r$.

The process of specialization incrementally and monotonically reduces the domain of the specialized function. The original (unspecialized) function definition has the largest possible domain, since any values could potentially be passed in at runtime. When specializing a function definition, the specializer uses the information in the argument specification to perform a reduction if that reduction is valid for all instances of the specification. Some reductions (such as reducing `(+ 1 2)` to `3`, or reducing `(if #t 'foo 'bar)` to `foo`, where the `1`, `2`, `#t`, `foo`, and `bar` are constants in the program) are always valid, regardless of the specification, and do not reduce the domain of the specialization. Others, however, that rely on information in the specification (such as reducing `(number? x)` to `#t` when `x` is specified as $\top_{Num}$, or reducing `(+ y 1)` to `3` when `y` is specified as `2`) reduce the domain of the specialized function.

Our approach relies on maintaining, at all times, an approximation of the most general index of each function currently being specialized. Just as argument specifications are comprised of value specifications, one per argument, the approximation of the MGI is maintained on a per-argument basis. For each argument to a specialization, the corresponding portion of the approximation starts out at $\top_{Val}$; each time the specializer performs a reduction that reduces the domain of the specialization, it updates the approximation. When the specializer is finished building the specialization, the approximation will be correct, and can be cached along with the specialization and the index.

We implement this in FUSE by adding a new attribute, the *domain specification*, to all symbolic values. Whenever FUSE decides to specialize a function on some argument specification, it makes copies of all of the symbolic values in the specification, and sets the domain specification of each copy to `top`. As it constructs the body of the specialization (by applying the body to the new, copied argument specification), it uses the value specifications to perform various reductions. Whenever a reduction reduces the DOS of the specialization, the corresponding domain specification is side-effected to a value lower in the lattice (though never lower than the corresponding argument specification; for any symbolic value with argument specification $s$ and domain specification $d$, we require that $s \sqsubseteq d$). This "lowering" can happen in one of three cases:

- *conditionals:* When the test of a conditional is known, the conditional is reduced to one of its two branches, and the domain specification of the test is set to the value specification of the test. If the test is unknown, its domain specification is left unchanged.

- *primitives:* When a primitive is reduced, any information it uses about its arguments must be reflected in the domain specifications of those arguments. That is, if a primitive $p$ is reduced on an argument with specification $s$, it must set the argument's domain specification, $d$, to a value sufficiently low in the lattice that $\forall x \in C[d](E(SP(p,d),x) = E(SP(p,s),x) = E(p,x))$. Furthermore, it should never be the case that reducing a primitive on a specific argument specification should use more information than reducing it on a more general argument specification; that is, if reducing $p$ on $s$ sets the argument's domain specification to $d$, and reducing it on $s'$ sets it to $d'$, then $s \sqsubseteq s' \Rightarrow d \sqsubseteq d'$.

  For instance, executing `number?` on a symbolic value denoting `6` reduces to `#t`; this used the fact that `6` is a number, so its domain specification is lowered to `top-number`. Executing `1+` on a symbolic value denoting `6` returns `7`; in this case, the `6` was used for its value, and its domain specification is lowered to `(number 6)`.

  Note that this strategy occasionally loses accuracy due to the limitations of the FUSE type system. One example is reducing `(null? x)` to `#f` when `x` is known to be the pair `(1 . 2)`. The information being used is that `x` is not the empty list; but there is no way of expressing this in the type system. The best we can do is to find the highest point in the lattice which lies above `x` but not above the empty list;[4] in this case, it is `(pair (`$y$ `.` $z$`))`, where $y$ and $z$ are symbolic values with value specifications `top`. This gives a conservative estimation of the DOS, because we are restricting it to all pairs rather than all objects other than the empty list. A similar case is reducing `(+ a b)` to `3` when `a` is known to be `1` and `b` is known to be `2`. Ideally, this should restrict the domain to those cases where the sum of `a` and `b` is `3`, but since we cannot express that, we instead restrict the domain to one where `a` is exactly `1` and `b` is exactly `2`; in this case, we do no better than normal polyvariant specialization. A better type system capable of expressing negation would be helpful here.

- *function application:* When the head of the application is known, its domain specification is set to its value specification. After this is done, if the head is a primitive, the specializer simply applies it (computing a new symbolic value for the returned value, which may include residual code if the primitive cannot be reduced), while if it is a user procedure, the specializer must make the usual unfold/specialize decision. If the decision is to unfold, the body is evaluated on the actual parameter values in the usual way; any conditionals, primitives, or function applications in the body will alter the domain specifications of the actual parameters to the unfolded procedure. If the decision is to specialize, the specializer will either re-use a previous specialization, or construct a new one (how this choice is made is described in the next section). After this choice is made, the domain specification of each actual parameter

[4] This method presumes that a unique such point exists; this happens to be the case for FUSE's type lattice, because $\perp_{Val}$ is the only type with multiple parents. More complicated type lattices would require a different mechanism for expressing the complement of a type.

```
;;; This figure is a transcript; lines beginning with "=>" were typed by the user

=> (define test
    '(lambda (x y z)
             (if x
                 (+ 1 z)
                 (* 10 (car y)))))

;;; Note: For brevity, pp-graph prints specifications using a representation
;;; slightly different from that described in the text and shown in Figure 2.
;;; Concrete values print as themselves, and pairs print in standard Scheme fashion.
=> (pp-graph (sp test #t '(2 . 3) (a-value)))
(((no-name                          ; name of specialization
   ((#T (2 . 3) top)                ; index (value attributes only)
    (#T top top)))                  ; MGI (valid for all y)
  (lambda (x3 y2 z1) (+ 1 z1))))    ; body of specialization

=> (pp-graph (sp test (a-value) '(2 . 3) (a-value)))
(((no-name
   ((top (2 . 3) top)               ; index
    (top (2 . top) top)))           ; MGI (didn't use cdr of y)
  (lambda (x3 y2 z1)                ; body
    (if x3 (+ 1 z1) 20))))

=> (pp-graph (sp test (a-value) `(,(a-number) . 3) (a-value)))
(((no-name
   ((top (top-number . 3) top)      ; index
    (top (top-number . top) top)))  ; MGI (did use type of car of y)
  (lambda (x3 y2 z1)                ; body
    (if x3
        (+ 1 z1)
        (tc-* 10 (tc-car y2))))))   ; tc-* is * without runtime type checks
```

Figure 3: Examples of specializations and their MGIs

must be lowered to indicate that the information in the arguments was used to choose a particular specialization. We do this by setting the the domain specification of each actual to the greatest lower bound of the actual's domain specification and the corresponding element of the chosen specialization's MGI.

Once a specialization is complete, the domain specification attributes of the symbolic values in the index of the specialization form the MGI for that specialization. The specializer caches this approximation along with the specialization index and the specialized function body, so that it can be used when making re-use decisions for that specialization. For an example of MGI computations, see Figure 3.

### 2.3 Using the Approximations to Control Re-Use

We have shown how the specializer computes an approximation to the domain of specialization of each specialization it builds; we now turn our attention to using this information to control re-use of specializations.

As stated above (Definition 3), having the DOS allows us to decide when it is safe to re-use a specialization. Since the DOS is not computable, we will use its approximation, the MGI instead, substituting domain operators for the set operators in Definitions 3 and 5. Given a specialization $r$ of a function $f$, at any particular call site of $f$ on arguments $a$, if the value specification of $a$ is less than the MGI of $r$, then it is safe, though not necessarily desirable, to re-use $r$.

Definition 5 provides one method of determining whether to compute a new specialization or re-use an existing one. It suggests building a new specialization, then comparing the domains (actually, the best we can do is to compare the MGIs, which are approximate specifications of the domains) of the two specializations: if the domain of the new specialization is smaller, use it; otherwise, discard it and re-use the old specialization. This technique works, and FUSE sometimes is forced to use it. Unfortunately, it has the disadvantage that it often computes specializations only to find that they are redundant. This technique will therefore produce better residual programs, but will not save effort at partial evaluation time.

A somewhat more efficient technique exists: we can often prove that the domains of the old and new specializations will be the same without building the new specialization. Assume function $f$ has been specialized on an index (value specification) $v$, producing specialization $r$ with MGI (domain specification) $d$. Specializing $f$ on another value specification $v'$ where $v \sqsubseteq v' \sqsubseteq d$ will yield a copy of $r$. Intuitively, $v'$ contains enough information to allow us to re-use $r$ safely, but no new information which could make building a new specialization worthwhile. Consider the cases:

1. ($v' \not\sqsubseteq d$). In this case, since the specialization $r$ is guaranteed to be valid only for values denoted by $d$, and $v'$ may denote values not denoted by $d$, it is not safe to re-use $r$. Therefore, build a new specialization on $v'$.

2. $v \sqsubseteq v' \sqsubseteq d$. In this case, $v'$ denotes a subset of the denotation of $d$, so it is safe to re-use the specialization. Given $v$, only the information in $d$ was actually used in building $r$, and $v'$ contains less information than $v$, so we have nothing to gain by building a new specialization (which would necessarily have MGI $d$). Therefore, re-use $r$.

3. otherwise. In this case, $v'$ denotes a set of values which is not a superset of those denoted by $v$. Even though it is safe to re-use $r$, building a specialization on $v'$ might be worthwhile.

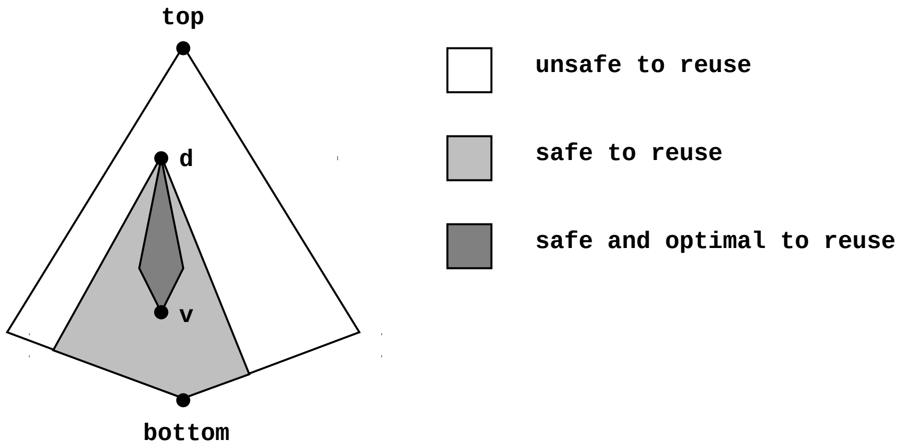


Figure 4: Basic re-use criterion. Given an argument specification $v'$, the reusability of an existing specialization with argument specification $v$ can be determined by locating $v'$ in the diagram above. Re-use is safe when $v' \sqsubseteq d$ and is optimal when $v \sqsubseteq v' \sqsubseteq d$.

---

Therefore, build a new specialization $n$; if it is more specialized than $r$ (by the criterion of Definition 4), then use it, otherwise discard it and re-use $r$. In the latter case, discard only the body of the new specialization $n$; a relation between the index of $n$ (namely, $v'$) and the MGI and body of the old specialization $r$ is still cached. Thus the knowledge that $r$ can be re-used for specification $v'$ is not lost, saving the specializer from performing this needless computation again (we will improve this portion of the criterion in Section 3.3.1).

A diagram of these cases is shown in Figure 4. Case 2 is important because it allows the specializer to avoid redundant work.

Before presenting FUSE's algorithm for re-using specializations, we must cover one more technical point dealing with recursion in specializations. FUSE specializes functions in a depth-first manner; during the specialization of a function $f$, it may specialize other functions that $f$ calls. Recursion arises when $f$, directly or indirectly, calls itself. When the specializer reaches a recursive call to $f$, it has a problem, because the decision whether to re-use the specialization of $f$ currently being built depends on the MGI of this specialization, which hasn't yet been fully computed. The specializer cannot use the partially-computed MGI, because the final MGI might be more restrictive, and thus the recursive call might be an unsafe one (*i.e.* the specialization of $f$ might end up being for a domain that doesn't include the values being passed to the recursive call). Instead, when deciding whether to re-use a specialization that is currently being built, FUSE uses the specialization index (which is guaranteed to be less than or equal to the final MGI) as the MGI. This strategy will occasionally cause the specializer to build a redundant specialization of $f$, which will then be detected and removed when both specializations of $f$ are complete. Such cases are quite rare in practice.[5] One might consider postponing the re-use decision rather than using an inaccu-

[5]Many online specializers, including some versions of FUSE, use a termination criterion which forces all non-

```
specialize(fcn, args, cache)
  for each specialization r of fcn in cache
    let the_mgi = if in_progress(r) then index(r) else mgi(r) endif
    if (index(r) <= args) and (args <= the_mgi)
    then return r
    else
      ; n is the cache object representing the new specialization
      let n = <parent=fcn, index=copy of args, mgi=empty, body=empty>
      set domain specifications of all symbolic values in index to top_val
      add n to cache
      build_specialization(n) ; side effects n and cache
      for each specialization r<>n in cache
         if not in_progress(r) and (n.mgi = r.mgi)
         then set n.body = r.body; return r endif
      return n
    endif

build_specialization(n)
   let body = n.parent specialized on n.args
              ; unfolds parent on arguments
              ; makes recursive calls to specialize if necessary
              ; computes mgi(n) in domain specification slots of n.args
   set n.body = body
   set n.mgi = collect domain_specifications of n.args
```

Figure 5: FUSE's algorithm for re-using specializations

rate MGI (the argument specification). Katz [28] has proposed such a mechanism (in his case, he needs the more accurate MGI information because he uses it to make the specializer terminate).

FUSE's algorithm for re-using specializations is shown in Figure 5. As an example of its usage, consider the function definition

```
(define test
  '(lambda (x y z)
     (letrec
         ((f (lambda (n l)
               (if (null? l)
                   '()
                   (cons
                    (if (number? n)
                        (+ (car l) 1)
                        (car l))
                    (f n (cdr l)))))))
       (let* ((t1 (f x z))
              (t2 (f y z)))
         (cons (t1 t2)))))).
```

Using

```
(sp test 1 (a-number) (a-value))
```

will specialize this function definition on `x=1`, `y` a number, and `z` unknown. The specializer first builds a specialization of `f` for `n` specified as `(number 1)`. However, the value `1` is never used, and thus `n`'s entry in the MGI will be `top-number`. When the specializer specializes the second call to `f`, where `n` is bound to `top-number`, it finds that the existing specialization satisfies the re-use criterion, and reuses it. The residual program is thus

```
(((no-name                              ; name
   ((1 top-number top)                  ; index
    (top-number top-number top)))       ; MGI
  (lambda (x3 y2 z1)                    ; body
    (cons (f6.1 x3 z1) (f6.1 y2 z1))))
  ((f6.1                                ; name
    ((1 top)                            ; index
     (top-number top)))                 ; MGI
   (lambda (n5 l4)                      ; body
     (if (null? l4) '()
         (cons (+ (car l4) 1)
               (f6.1 n5 (cdr l4))))))).
```

Running the command

```
(sp test (a-number) 1 (a-value))
```

unfoldable recursive calls to a function $f$ within a specialization of $f$ to invoke that same specialization, generalizing the arguments as necessary to make this possible. In such cases, using the specialization index when examining incomplete specializations does not risk building a redundant specialization.

to invoke the specializer produces a different behavior. In this case, the first specialization built is for `n` specified as `top-number`. This type information is used, and thus the MGI also specifies `n` as `top-number`. When the second call to `f` is specialized, the value of `n`, namely `(number 1)`, is more specific than the index of the existing specialization, so the re-use criterion fails, and a new specialization is built. However, in this new specialization `n` is used only for its type and not for its value, so the MGI specifies `n` as `top-number`. At this point, the specializer finds that it has already built a specialization with this same MGI, and re-uses it, discarding the new specialization, giving the following residual program:

```
(((no-name                              ; name
   ((top-number 1 top)                  ; index
    (top-number top-number top)))       ; MGI
  (lambda (x3 y2 z1)                    ; body
    (cons (f6.1 x3 z1) (f6.1 y2 z1))))
  ((f6.1                                ; name
    ((top-number top)                   ; index
     (top-number top)))                 ; MGI
   (lambda (n5 l4)                      ; body
     (if (null? l4) '()
         (cons (+ (car l4) 1)
               (f6.1 n5 (cdr l4))))))).
```

Thus, in this case, FUSE's strategy produces the desired residual program, but does not save work at specialization time. Section 3.3.1 describes an improvement which avoids such redundant work by keeping track of which information was demanded but unavailable; in this case, it would deduce that the numerical value of `n` was not demanded, and would not build and discard the specialization for `n` as `1`.

# 3 Extensions

This section describes three classes of extensions to the basic re-use algorithm given above. The first class of extensions is necessary to make the re-use mechanism compatible with the first-order information preservation mechanisms of [49, 39]. The second subsection describes how our mechanism interacts with higher-order specialization as implemented in FUSE, and with the higher-order specialization techniques described in [39]. Finally, we describe two extensions which improve the efficiency of specialization by avoiding the construction of specializations which are sure to be discarded, and by limiting the use of information during the construction of specializations. These extensions are, for the most part, orthogonal, and do not interfere with one another.[6]

These extensions will require that we modify the specializer. To aid in understanding the scope of these modifications, we separate them into three classes:

1. Modifications to the re-use criterion of Section 2.3. That is, change the inequality involving the arguments, specialization index, and MGI in the `specialize` procedure of Figure 5.

[6]One exception is that the first-order and higher-order mechanisms of [39] are mutually exclusive, so their re-use-related extensions need not be compatible.

2. Modifications to the computation of domain specifications (and thus MGIs), as described in Section 2.2. This may involve changes to the implementation of primitives, special forms, and function application in the specializer (*i.e.,* anything invoked by the `build-specialization` procedure of Figure 5).

3. Modifications to other operations in the specializer unrelated to specialization re-use. This includes the generalizer as described in [48] as well as the type inference mechanisms of [39].

When we describe modifications, we will attempt to categorize them using the above terminology.

## 3.1 Handling FUSE Type Inference

Unlike most other specializers, FUSE reasons about the values returned by residual conditional expressions and calls to specialized procedures, using the techniques of generalization and fixpoint iteration, respectively. Each of these techniques requires that we modify the re-use mechanism slightly. These modifications will not affect the re-use criterion, but will affect both the computation of domain specifications and the operation of the type-inferencing portion of the specializer.

### 3.1.1 Residual Conditionals

FUSE is often able to compute useful information about the values returned from `if`-expressions with unknown tests, by returning the generalization of the value specifications of the two arms of the conditional [48]. For instance, specializing

```
(lambda (p)
  (number?
    (if p
        1
        2)))
```

on completely unknown `p` yields the original function definition (modulo renaming) under most specializers, but instead yields

```
(lambda (p x y z)
  #t)
```

under FUSE.[7] This mechanism for propagating information out of residual code necessitates a minor modification to our procedure for computing the MGI, as described above Section 2.2. Consider specializing

```
(lambda (p a b)
  (let ((c (if p a b)))
    (cons c (number? c))))
```

[7]For more realistic examples of the usefulness of such generalization, see [39, 49]

where p is unknown, a=1 and b=2. The result of (if p a b) (bound to c) will be a symbolic value with value specification top-number and domain specification top. Applying the primitive number? will make use of the fact that its input is a number, and will lower the domain specification of (the symbolic value bound to) c to top-number. Unfortunately, the domain specifications for (the symbolic values bound to) a and b were never changed; therefore, the MGI for the the specialization will be (top top top) when it should be (top top-number top-number).

The solution is simple: whenever the domain specification of a symbolic value obtained from a generalization is lowered, the domain specifications of both inputs to the generalization must also be lowered accordingly. Therefore, we add a parents attribute to all symbolic values; whenever a generalization is performed, the parents field of the output symbolic value is set to point to both of the input symbolic values. All side effects to the domain specification slot of a symbolic value are propagated, recursively, to its parents. Thus, in the example above, the result of evaluating (if p a b) returns a symbolic value $s$ with value specification top-number, domain specification top, and parents ($a$ $b$), where $a$ and $b$ are the symbolic values bound to a and b, respectively. When the number? operator sets the domain specification of its input, $s$, to top-number, it also sets the domain specifications of $s$'s parents, $a$ and $b$, to top-number, and the correct MGI is computed.

This extension is a type 2 modification; it affects only the computation of domain specifications.

Further complications arise because of FUSE's ability to return information out of calls to specialized functions. Most specializers simply treat the output of such a residual function call as though it were $\top_{Val}$. However, FUSE is different; for each specialization it builds, FUSE computes a generalized return value specification, which specifies those values that could be returned from *any* call to this specialized function. This information can then be used in specializing the function containing the residual call (for examples, see [49]). The generalized return value specification for a given specialization is computed by assuming that the return value is $\perp_{Val}$, and performing fixed point iteration until the return value converges [49, 39].

This mechanism conflicts with the specialization re-use analysis because the return value specification is computed during specialization, and is based on the specialization's index, not its MGI, and is thus valid only at the original call site, not at subsequent call sites. For example, specializing the function

```
(lambda (x y z)
  (if x
      y
      z))
```

on x specified as #t, y specified as top-number, and z specified as top yields

```
(lambda (x y z)
  y)
```

a return value specification of top-number, and a MGI in which y is specified as top. Thus, we can safely and optimally re-use the specialization on a different set of parameters where y has any specification that lies below top but not below top-number. Note, however, that the return value specification is in error; although the specialization is indeed applicable to y specified as top, the return value specification for that case would be top, not top-number. This occurs because the return value specification is built using the *value specifications* of the arguments the function was specialized on, even though the generated code may be valid for more general inputs (up to and including the MGI).

We must choose between computing a single, general, return value specification that is valid for all calls to the specialization, or computing a different one for each call site. Computing a single value is straightforward; we must ensure that only information used in the building of the specialization is used in computing the return value specification; values that are not used may not constrain the return value. This has the disadvantage that accuracy is lost, since all calls to a specialization must share a common return value specification.

Computing a different return value per call site is more attractive, since it preserves more information. However, since FUSE computes return value specifications by computing the body of the specialization and taking its value specification attribute, doing this would seem to require recomputing the body of the specialization on each set of actual parameter specifications. Luckily, we need not perform such a recomputation. At runtime, the value (or its subparts, if it is a data structure) returned by a call to a user procedure may come from one of three places. A returned value can be:

1. a constant in the procedure (*i.e.,* `4` or `"foo"`), or

2. constructed by the procedure (*i.e.,* the result of `cons`, `+`, or `lambda`), or

3. all or part of one of the procedure's arguments (*i.e.,* `(car <formal>)`).

These are the only possibilities. Constants are not a problem, as they will be the same for every call site. Values constructed by the procedure are also valid for every call site, since computing them alters the domain specifications of their arguments, thus restricting the MGI of the specialization such that only call sites that would construct the same values will re-use this specialization. This leaves the case of argument values that are "passed through," which can be solved by changing the interpretation of the return value specification.

Instead of treating the return value specification as a value, and using it as the return value specification for all call sites, we will instead treat it as a template to be instantiated at each call site. When the decision to re-use a specialization is made, the specializer matches the symbolic values in the arguments against those in the specialization index, and builds a substitution environment. Any part of the specialization index appearing in the template (which was built from the specialization index during the specialization process) is replaced by the corresponding part of the argument specification. There is a technicality related to generalization: if a value in the return value approximation was obtained by generalizing two other values, then the generalization must be performed again after substitution has taken place. This requires that the specializer keep track of how generalized values were computed; the `parents` attribute used for handling residual conditionals can be used to do this. If a return value has symbolic values in its `parents` attribute, we instantiate them, then generalize the instantiated values. Consider specializing the function

```
(lambda (a b c d)
  (list 44 (+ a 1) c (if b c d)))
```

on `a=2`, `b` unknown, `c=(foo 1)` and `d=(foo 2)`[8]. We will use the names $a$, $b$, $c$, and $d$ to denote the symbolic values bound to the formal parameters `a`, `b`, `c`, and `d`, respectively. The specializer

[8] For simplicity, we use a notation which omits FUSE's manifest type specifiers. Technically, these specifications should be written as `(number 2)`, `top`, `(pair ((symbol foo) . (pair ((number 1) . nil))))`, and `(pair ((symbol foo) . (pair ((number 2) . nil))))`, respectively.

builds a specialization

```
(lambda (a b c d)
  (list 44 3 c (if b c d)))
```

and a return value template ($t_1$ $t_2$ $c$ $t_3$), where the value specifications of the symbolic values $t_1$, $t_2$, $c$, and $t_3$ are `44`, `3`, `(foo 1)`, and `(foo top-number)`, respectively. Additionally, the symbolic value $t_3$ has parents $\{c, d\}$. Thus, the return value specification for this call site, computed by taking the value specification attribute of all component symbolic values, is `(44 3 (foo 1) (foo top-number))`.

Assume that the specializer re-uses the specialization at another call site where `a` and `b` are unchanged, but `c=(foo 3)` and `d=(bar 3)`. We will use the names $a'$, $b'$, $c'$ and $d'$ to denote the symbolic values containing these value specifications. The substitution environment will bind the symbolic value the index corresponding to the formal parameters to their new values (*i.e.,* $\{a = a', b = b', c = c', d = d'\}$). Instantiation of the return template works as follows. The first two elements, $t_1$ and $t_2$, are not bound in the substitution environment, and are copied unchanged. The third element, $c$, is bound to $c'$, so that value is used. The fourth element, $t_3$, is not bound in the substitution environment, but was built via generalization. Looking up both of its `parents`, $c$ and $d$ in the substitution environment yields $c'$ and $d'$ respectively; generalizing their value specifications gives a new symbolic value $t_4$ with value specification `(top-symbol 3)`. Thus, the instantiated template is ($t_1$ $t_2$ $c'$ $t_4$), which has a value specification of `(44 3 (foo 3) (top-symbol 3))`, which is correct for the new parameter values.

This method allows the specializer to get as much information out of a call to re-used specialized procedure as it would have had it built a new specialization. Thus, it has the potential to reduce specialization time without compromising the accuracy of specialization. It is a type 3 modification; adding templates and instantiation affects only the type inference portion of the specializer, not the re-use mechanism.

## 3.2 Handling Higher-Order Programs

All of the examples thus far have been limited to first order programs. In this section, we show that our mechanism works equally well for higher-order programs, and describe the necessary changes to various higher-order specialization algorithms.

Specializers represent families of runtime closures by single closures at specialization time. These closure approximations contain an expression and an environment binding free variables to specialization-time approximations of runtime values. Closures are first-class objects and may be passed anywhere; if two closures of the same `lambda`-expression are generalized, corresponding slots in the environment are generalized also. Handling higher-order programs requires that we produce accurate use information for values captured in closures, as well as values passed as arguments to closures. Since building a closure is a "non-touching" operation, it does not affect the MGIs of any values in "closed-over" variables; all of the MGI-lowering work happens when closures are unfolded or specialized. We treat unfolding first, then specialization.

### 3.2.1 Unfolding Closures

Unfolding a closure is similar to unfolding a first order procedure, in that the domain specifications of the actual parameters may be altered if they are used in the body of the closure. In addition, the

```
(define (kons the-kar the-kdr)
  (lambda (msg)
    (case msg
      ((kar) the-kar)
      ((kdr) the-cdr))))

(define (kar the-kons)
  (the-kons 'kar))

(define (kdr the-kons)
  (the-kons 'kdr))
```

Figure 6: An implementation of pairs using closures and message passing

domain specifications of values in the closure's environment (*i.e.,* values bound to its free variables) may also be altered. Both cases are treated identically; that is, whenever a value is used to perform a reduction, its domain specification is altered. Because unfolding the closure makes use of the closure's body and environment, the domain specification of the closure is lowered accordingly.

For example, consider an implementation of pairs using closures (Figure 6). If we specialize

```
(define (test x y)
  (+ (kar x) y))
```

on `x=(kons 1 2)` and `y` unknown, and unfold both the application of the procedure `kar` and the closure bound to `x`, the residual code will be

```
(define (res-test x y)
  (+ 1 y))
```

where the domain specification for `x` is a closure consisting of the expression `(lambda (msg) ...)` and bindings for the free variables `the-kar` (bound to `1`) and `the-kdr` (bound to `2`), and where the domain specification for `a` is `top`. The symbolic values in the closure's environment also have domain specifications; `the-kar` has specification `(number 1)`, while `the-kdr` has specification `top`. Thus, the specialization `res-test` could be re-used for `x=(kons 1 3)`.

Thus, we see that values in the environment of a closure which is unfolded are treated analogously to values contained in a data structure which is accessed, while values passed as actual parameters in an unfolded call to a closure are treated like actual parameters to a first-order procedure which is unfolded. This requires no modifications to the re-use criterion or to the specializer.

### 3.2.2 Specializing Closures

The previous subsection described how unfolded closures (which are completely consumed at specialization time) are treated, leaving open the question of how specialized closures (which appear in the residual program) are handled under our re-use mechanism. Specifically, we must decide when and how the domain specifications of actual parameter values in residual applications and free variables of residual closures should be modified.

The proper choices depend on how specializations of closures are constructed. Most existing specializers for higher-order Scheme [48, 11, 22, 4] build residual closures by specializing their bodies on completely unknown argument vectors. No knowledge of the argument values at the residual call sites of a closure is used in computing its specialization; thus, there is no need to lower the domain specifications of the actual parameters at any residual call site. The domain specifications of the free variables are also left untouched; because non-unfolded closure arguments to a procedure have no effect on the residual code for that procedure, it's never necessary to build distinct specializations of a procedure for call sites passing different closures unless those closures are unfolded within the specialization. Thus, no modifications are necessary.

The version of FUSE described in Section 4 of [39] and in [37] uses control flow analysis to compute more accurate argument vectors for specialized closures. We have not implemented specialization re-use under this algorithm, but believe that doing so would not be difficult; the remainder of this subsection describes how this could be done.

The control flow analysis approach still builds only one specialization per closure, but builds a more accurate specialization. The argument values at residual call sites of a specialized closure are indeed used in constructing the body of the specialized closure, but they do not affect the construction of the residual call, or any other code in the specialization containing the call. Thus, there is still no need to lower the domain specifications of the arguments at residual call sites. This differs from the construction of residual calls to specializations of first-order procedures, where we must lower the domain specifications of the actual parameter values to match those of the index of the specialization. The reason for the difference is that, in the first order case, the specializer uses the actual parameters to choose one of multiple specializations of the called procedure; different argument values could thus result in a different residual call. In the higher-order case, there is no specialization-time choice because there is still only one specialization per closure, and the determination of which specialization to invoke at a call site is not made until runtime.

However, even though the re-use mechanism doesn't have to be changed, the iterative algorithm of [39, 37] for computing the bodies of specialized closures does require a slight alteration to work properly in the presence of the re-use mechanism. The algorithm operates by incrementally determining, during the specialization process, which runtime call sites are reach by each closure. Each time a closure reaches a new call site, the argument specification of the specialized closure is recomputed by generalizing it with the argument specification of the call site; if the specialization's argument specification changes, the specialization is rebuilt. The correctness of this algorithm relies on the fact that the argument specification (*i.e.,* the vector obtained by taking the value specifications of each argument) for each call site specifies all values which could be passed to the specialized closure from that site at runtime, so that generalizing the specifications from all call sites will yield a specialization index which is sufficiently general. The re-use mechanism invalidates this assumption, because if a call site's enclosing specialization is re-used, different values (not reflected in the call site's argument specification) may reach the call site, but will not be seen by the iterative algorithm, resulting in an insufficiently general specialization of the closure.

Either of two possible modifications will solve this problem. First, the domain specifications of the arguments at a call site do correctly (though not necessarily precisely) approximate all values which could be passed at that site at runtime. Thus, we can simply modify the iterative algorithm to use domain specifications (once they are completely "lowered," which is true after the call site's enclosing specialization is complete) instead of argument specifications. Because the domain specifications at a call site approximate all values which might appear at that site as a result of any possible reuse of the call site's enclosing specialization, this approach can risk building an overly general specialization of a closure. The problem is that the domain specifications must represent all values which might arrive due to any legal re-use, not only those values which will actually arrive at runtime. For example, specializing

```
(define (g f x)
  (if (number? x)
      (f x)
      x))
```

on `f=top` and `x=4` will compute a domain specification of `top-number` for `x`, since the specialization can safely be re-used for any numeric `x`. Thus, the argument vector computed for any closure that might reach the call site `(f x)` will be at least as high as `top-number`. If the specialization of `g` is re-used on `x=3` and `x=5.5`, this will be optimal; however, if `g` is re-used only on integers, the call site `(f x)` should really only constrain the specializations of closures reaching the site to be specialized on `top-integer`.

We can solve this problem by not using the domain specifications when computing the argument specifications for specialized closures. Instead, each time a specialization is re-used, we must recompute the value specifications of all residual higher-order call sites within the body of the specialization, and use these specifications to recompute the argument specifications of any closures reaching the sites. This can be accomplished either by an instantiation procedure similar to that used for computing return values in Section 3.1.1, or by extending the incremental control flow analysis mechanism of [39, Section 4.5] to pass information about scalar values, in addition to pairs and closures, along initial source/final destination links. In any case, the modifications necessary to make the re-use mechanism work with higher-order specialization affect only the higher-order specialization mechanism, not the re-use mechanism (type 3 modification).

## 3.3 Improving Re-use

In the previous subsections, we have described how to extend the re-use mechanism to deal with type inference and higher-order extensions to the specializer. We now turn to the problem of improving the re-use mechanism itself, either to reduce the number of redundant specializations constructed and discarded, or to increase the number of opportunities for sharing through re-use.

### 3.3.1 Improving the Re-Use Criterion

As described in Section 2.3, our re-use criterion will re-use a prior specialization without first building a new one when it can prove that (1) it is safe to do so, and (2) building a new specialization would not be worthwhile. (1) is easily proved or disproved using the safety criterion of Definition 3, but (2) can be proved in only limited cases. The algorithm of Section 2.3 can prove (2) only when the new argument specification $v'$ is more general than the old one $v$ (*i.e.,* $v \sqsubseteq v'$). This is possible because more general values *always* lead to more general MGIs.

Unfortunately, this criterion is rather weak, and can cause the specializer to waste time building and discarding specializations. Two cases in which this occurs are as follows. First, a more specific argument value may not allow the specializer to build a better specialization: *i.e.,* specializing

```
(lambda (x) (number? x))
```

on 1 yields the same MGI as specializing it on `top-number`. If we build the specialization for 1 first, all is well, but if we build the one for `top-number` first, our algorithm has no way of knowing that 1 won't lead to any more reductions; it must evaluate `(number? 1)` in order to determine this, at which point it finds out it needn't have bothered. This is exactly the problem we saw in the second example on page 14. We call this the problem of *refined indices.*

A similar problem arises even when the new argument value is not more specific than the first one, but merely incomparable with it. Specializing the function above on 1 gives an MGI of `top-number`; on a subsequent attempt to specialize it on 2, the algorithm cannot determine that `number?` will treat 1 similarly to 2 without actually evaluating it. This can be a real problem in interpreters which pass explicitly tagged data structures, some of whose tags are never examined; since all of the tags are known symbols (incomparable in our type system), the specializer must build and discard specializations for each combination of tags in unexamined positions. As a trivial example, consider

```
(lambda (obj)
  (if (eq? (first obj) 'pair-tag)
      (second obj)
      'error))
```

specialized initially on `obj=(pair-tag (num-tag top) (num-tag top)))`[9] and subsequently on `(pair-tag (sym-tag top) (num-tag top))`. Both specializations use only the information that the argument, `obj`, is a pair whose `car` is the symbol `pair-tag`; thus, the MGI is `(pair-tag top . top))`. Unfortunately, because `sym-tag` and `num-tag` are incomparable, the $v \sqsubseteq v'$ test fails, and we must build the second specialization, only to find that it has the same MGI, and discard it. We will call this the problem of *incomparable indices.*

In both cases, our algorithm saves no work (*i.e.,* fails to prove that the new specialization will have the same MGI as the old one) because the MGI denotes does not denote just the set of values for it is *optimal* to re-use the specialization, but rather the entire set of values for which such use is *safe*. In this section, we will describe two ways of addressing these cases. Our first solution limits the behavior of primitive reductions in the specializer in order to solve the problem of incomparable indices, but doesn't address the problem of refined indices. Our second solution applies to both cases without limiting the behavior of the specializer, but requires that extra information be maintained.

The problem with incomparable indices arises when the specializer constructs a specialization on a specification $v$, resulting in MGI $d$, and is subsequently asked to compute a specialization on a specification $v' \sqsubseteq d$ where $v \not\sqsubseteq v'$ and $v' \not\sqsubseteq v$. If $v'$ denoted strictly less information than $v$, we would be able to establish that the new specialization could be no better than the prior one (*i.e.,* if we denote the new specialization's MGI by $d'$, $d' = d$), while if $v'$ denoted more information, the new specialization might be better (*i.e.,* $d' \sqsubseteq d$). Unfortunately, neither of these relationships holds here. The re-use algorithm cannot make any determination, and so must build a new specialization, then compare the two MGIs.

We solve this dilemma by requiring that specializations built on incomparable values have MGIs which are either equal or incomparable; we call this the *incomparability restriction.* Adopting this restriction makes the optimal re-use test decidable for all incomparable values. Consider building

[9] Once again, we are using an abbreviated notation in which concrete values and pairs are denoted in the usual Scheme fashion, and in which explicit type markers are omitted when they can be inferred. In this case, only `top` is a type marker. The full notation for this value is `obj=(pair ((symbol pair-tag) . (pair (symbol num-tag)...)))`

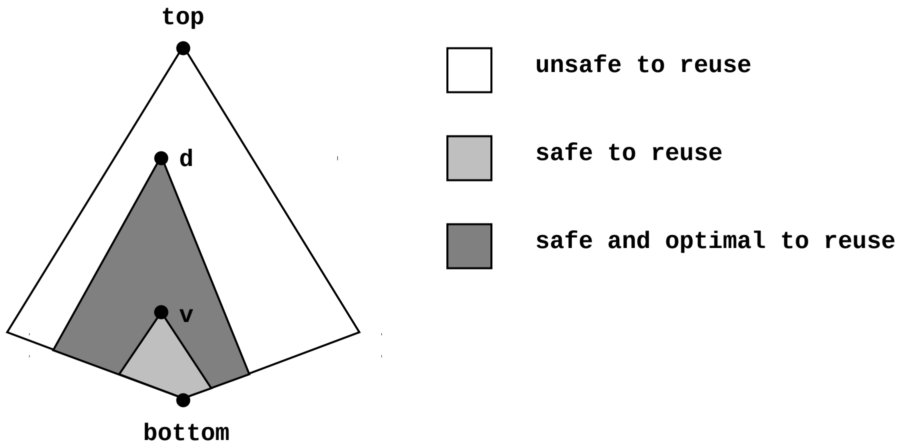


Figure 7: Re-Use criterion, extended to take advantage of the properties of incomparable indices. Given an argument specification $v'$, the reusability of an existing specialization with argument specification $v$ can be determined by locating $v'$ in the diagram above. Re-use is safe when $v' \sqsubseteq d$ and is optimal when $v' \not\sqsubseteq v$.

---

a new specialization on the new value $v'$. If $v' \not\sqsubseteq d$, re-use will be unsafe, and a new specialization must be constructed. If $v' \sqsubseteq d$, then, by the properties of specialization, we know that $d' \sqsubseteq d$ (where $d'$ is the MGI of the new specialization on $v'$). Adding the incomparability restriction tells us $d' = d \vee (d' \not\sqsubseteq d \wedge d \not\sqsubseteq d')$; combining these properties gives $d' = d$, so we can conclude that the new specialization will be redundant. Thus, for incomparable values $v$ and $v'$, re-use is always either unsafe or optimal; we will never have to build a specialization for $v'$ and discard it.

This allows us to change the re-use criterion to re-use the existing specialization whenever $v' \sqsubseteq d \wedge v' \not\sqsubseteq v$ (for a pictorial representation of this criterion, see Figure 7).[10] For example, a specialization of

```
(lambda (x) (number? x))
```

built on `x=1` (with MGI `top-number`) could be re-used for `x=2`, because $2 \sqsubseteq \top_{Num} \wedge 2 \not\sqsubseteq 1$. Similarly, applications of

```
(lambda (obj)
  (if (eq? (first obj) 'pair-tag)
      (second obj)
      'error))
```

with different values for `(second obj)` would share the same specialization. This criterion is not useful for refined indices; *e.g.,* a specialization of

[10] This is the solution originally used in FUSE, and documented in the original version of [36]. The revised version of that paper [35] switched to the criterion of Section 2.3, which is applicable to all specializers, not just FUSE.

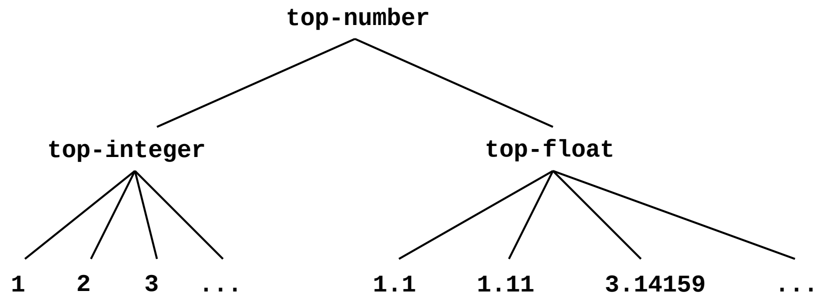


Figure 8: A subdomain for numeric types

```
(lambda (x) (number? x))
```

built on `x=top-number` could not be re-used for `x=1`, because $1 \sqsubset \top_{Num}$.

The question at hand is whether the incomparability restriction constrains the specializer too strongly. Basically, it forces applications of primitive operators in the specializer to treat siblings in the domain of values consistently: it must never be the case that one subtype of a type is used completely, while another is not used at all. For example, in the numeric subdomain of Figure 8, any primitive which uses an argument of `top-int` (*i.e.,* lowers its domain specification to `top-int`) would also have to use an argument of `top-float`. An argument of `(float 5.5)` wouldn't have to be completely used, but its domain specification would have to be lowered to at least `top-float`. For example, the predicate `integer?` has the desired behavior (it uses the fact that its input either is or is not an integer). Conversely, any primitive which doesn't make use of all of the information in a particular value must not make use of all of the information in any incomparable argument value (*e.g.,* since `number?` uses only `top-number` when applied to `(integer 5)`, it must also use `top-number` when applied to `(integer 6)`, `top-float`, or `(float 5.5)`).

This property holds for most reasonable implementations of specialization-time primitives. We have encountered only one optimization which violates this property: algebraic simplification. Consider an implementation of `+` which reduces to one of its arguments when the other argument is is known to be `0`. Applying `+` to `(integer 1)` and `top-integer` yields an MGI of `(top-integer top-integer)`[11], while applying it to `(integer 0)` and `top-integer` yields an MGI of `((integer 0) top-integer)`. This is not a problem in the current incarnation of FUSE, which does not perform such simplifications; specializers with such optimizations could not use this approach.[12]

[11]This is assuming that we are limiting specialization by forbidding the inlining of the constant `1` in this case; for details, see the discussion of limiting in Section 3.3.2.

[12]Actually, this criterion works even in the presence of algebraic simplifications, provided that a sufficiently detailed type lattice is used. Consider a case where, in addition to lattice elements for $\top_{Int}$ and the concrete integers 1, 2, etc., the lattice contained an element $\top_{nonzero\text{-}integer}$. Then applying `+` to `(integer 1)` and `top-integer` would

The incompatibility restriction is both a type 1 modification, in that it modifies the re-use criterion, and a restriction on the operation of the specializer (*i.e.*, it cannot violate the incompatibility restriction), which might require a type 3 modification to specializers which don't already have the incompatibility restriction property.

A second approach, which addresses the problem of refined values in addition to that of incomparable values, uses a different tactic. Instead of constraining the specializer (via the incomparability restriction) so that re-use decisions can be made using the argument values and the MGI alone, this approach computes additional information which allows the specializer to prove that re-use of a specialization is optimal. Remember that we want to know whether, given a specialization constructed on argument specification $v$, yielding MGI $d$, building a new specialization on $v'$ where $v' \sqsubseteq d$ might be worthwhile. It isn't worthwhile for $v \sqsubseteq v' \sqsubseteq d$ (*c.f.* Section 2.3), but we need a solution for the cases $v' \sqsubseteq v$ and $v \not\sqsubseteq v'$. We accomplish this by keeping track of not only the information used in constructing a specialization, but also what (unavailable) information would allow a better specialization to be constructed. For example, specializing

```
(lambda (x) (number? x))
```

on `top-number` yields an MGI of `top-number`, and notes that no amount of additional information would be useful (in building a new specialization). Thus, building a new specialization for `(number 1)` would be a waste of time. Similarly, specializing

```
(lambda (x) (+ x 5))
```

on `top-integer` yields an MGI of `top-integer`, and notes that any value $v$ for `x` where $v \sqsubset \top_{Int}$ would lead to a better specialization. Thus, new specializations would be built for `x=1`, `x=2`, etc.

There are several possibilities for representing this notion of useful additional information. One might consider computing a set of all values which would be useful; this is unrealistic because this set would often be infinite (in the previous example, all integers are useful). A more practical method approximates the set of useful values by its least upper bound in the lattice; the idea is that any value lying at or below this bound is potentially useful, while any value lying above this bound is guaranteed to yield the same MGI as that of the specialization associated with the bound.[13] We will call this bound the "demand specification," as it represents information that is demanded but unavailable. Given a specialization built on value $v$ with MGI $d$ and demand specification $u$, we can avoid building a new specialization for another argument value whenever re-use is safe ($v' \sqsubseteq d$), and either

- The new value is more general than the old one ($v' \sqsupseteq v$), or
- The new value is not more specific than the most general "useful" value ($v' \not\sqsubseteq u$).

This criterion is depicted pictorially in Figure 9. For example, specializing

give an MGI of `(top-nonzero-integer top-integer)`, while applying `+` to `(integer 0)` and `top-integer` would yield an MGI of `((integer 0) top-integer)`. Because `top-nonzero-integer` and `(integer 0)` are incomparable, the MGIs are incomparable, and the incomparability restriction holds.

[13] This is still somewhat coarse, as there is no way to say that all known integers are useful without also saying than `top-integer` is useful also (though, if, as in the example, `top-integer` is the MGI, we know it isn't useful). Maintaining such a distinction would require the use of a domain that models sets of values taken from the specializer's value domain, including partially known elements like `top-integer`.

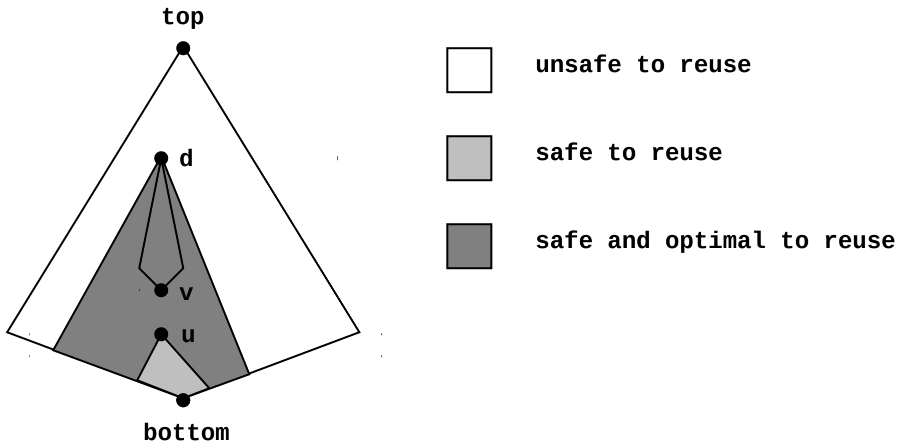


Figure 9: Re-Use criterion, extended with an approximation of "useful" information. Given an argument specification $v'$, the reusability of an existing specialization $v$ can be determined by locating $v'$ in the diagram above. Re-use is safe when $v' \sqsubset d$ and is optimal when $v \sqsubseteq v'$ or $v' \not\sqsubseteq u$. The graphic above depicts a case where $u \sqsubset v$; this is not necessarily the case.

---

```
(lambda (x) (number? x))
```

on either `top-number` or `1` would yield an MGI of `top-number` and a demand specification of `bottom`, indicating that all non-bottom values which are numbers will result in the same specialization. Thus, unlike the criterion of Figure 7, we obtain the desired result (re-use of the specialization) for both refined indices (first specializing on `top-number`, then on `2`), and for incomparable indices (first specializing on `1`, then on `2`). Furthermore, this criterion operates correctly under specializers that perform algebraic optimizations. Specializing

```
(lambda (x y) (+ x y))
```

on `x=1` and `y=top-integer` yields the specialization

```
(lambda (x y) (int-+ x y))
```

which has a MGI of `(top-integer top-integer)` and a demand specification of `(top-integer top-integer)`. Thus, the specialization can be re-used for `x=top-integer`, `y=top-integer` (because only the fact that both `x` and `y` were integers was used in building the specialization), but not for `x=0`, `y=top-integer`, because that argument vector would yield a different specialization, namely

```
(lambda (x y) y).
```

We can simplify our implementation somewhat by noting that, in practice, it is almost always the case that the demand approximation is either equal to the MGI (meaning that values below the MGI would be useful) or bottom (meaning that all values below the MGI are useless, except for bottom, which is always useful for strict primitives). Cases where this is not true do exist; for example, siblings in the type lattice may not be equivalently useful in performing reductions. Consider a lattice which contains $\top_{Num}$, $\top_{Fixnum}$, and $\top_{Bignum}$ (where $\top_{Fixnum} \sqsubset \top_{Num}$ and $\top_{Bignum} \sqsubset \top_{Num}$) and and an implementation of `1+` which can make use of the fact that its argument is a fixnum, but not that it is a bignum. Applying such an operator to `top-bignum` would return a domain specification of `top-number` and a demand specification of `top-fixnum`. However, we have yet to encounter any such cases in our partial evaluators.

This allows the simplification of reducing the demand specification to a single boolean flag per symbolic value. If the flag is true, then the demand specification is the same as the domain specification (*i.e.,* values below the domain specification but above bottom are useful); otherwise, the demand specification is `bottom` (the only useful value below the domain specification is the bottom element). Primitives which receive an argument which is sufficiently known to allow them to be reduced completely (*e.g.,* `number?` applied to `top-number`) set the flag to false; otherwise, the flag is set to true. The re-use criterion under this simplification is as follows. Given a specialization built on value $v$ with MGI $d$, and with demand function $f$ mapping a symbolic value to its demand flag, we can avoid building a new specialization for another argument value whenever re-use is safe ($v' \sqsubseteq d$), and either

- The new value is more general than the old one ($v' \sqsupseteq v$), or
- The specialization is optimal for all non-bottom values, and the new value is not bottom (for all subparts $x$ of $v'$, $\neg f(x) \wedge (x \neq \perp)$).

This criterion allows us to handle the second example of page 14 without having to build and then discard a specialization built on `n=1`.

Computing "useful" information is both a type 1 and a type 2 modification; not only do we change the re-use criterion, but we must compute new information (demand specifications) to evaluate the criterion.

### 3.3.2 Limiting Specialization

In the previous subsection, we increased the amount of re-use by improving the accuracy of the re-use criterion, allowing it to avoid constructing new specializations in a greater number of cases. In this subsection, we describe another way to achieve more re-use, not by improving the estimation of the specializer's use of information, but by limiting that use;*i.e.,* constructing more general specializations.

The re-use analysis described above operates by determining which of the information present in the original argument specification (specialization index) is used, and re-using the specialization on argument specifications for which it can prove that the same information will be used. What should "used" mean? We have been using the criterion that information about a value is used when it allows the specializer to perform a reduction at specialization time that otherwise would have to be performed at runtime. Using information in a value specification to decide a conditional, apply a function call head, or execute a primitive procedure are all clearly instances of this criterion. What is more difficult to decide is what to do when given information about a primitive procedure's arguments, but not enough to execute the primitive. Such an example is specializing

```
(lambda (x y)
  (+ x y))
```

with `x` specified as `(number 2)` and `y` specified as `top`. Our choice is between producing the specialization `(lambda (x y) (+ 2 y))`, thus restricting the domain to `x` equal to `2`, or producing `(lambda (x y) (+ x y))`, which doesn't restrict its domain. Both specializations return the same information (namely, that the result of applying it is a number), and neither one completely reduces the application of `+`. The question is: "Is the first specialization better?" In our view, this depends on the virtual machine that will be executing the residual program. Some processors have an "add immediate" instruction, which can add the constant 2 to a register value faster than it can add two register values, some take the same amount of time for both operations, and some might even run more slowly on the first specialization because of the overhead of loading the value `2` into a register. Also, building residual code of the form `(+ 2 x)` makes fewer specializations re-usable, increasing the size (and slowing the performance) of residual programs. One way out of the problem would be to delegate it to the code generation postpass, which could be expected to know more about the target architecture; it could detect and merge specializations which it considered to be the same. However, this would remove an opportunity for speeding up the specializer; since, if the specializer knew these rules, it could avoid building redundant specializations such as `(lambda (x y) (+ 2 y))` and `(lambda (x y) (+ 3 y))`, instead of building both and having the code generator "clean up" later.

Another opportunity for sharing comes from *non-touching* constructor primitives such as `cons`, which can be reduced[14] at specialization time without any knowledge about its arguments. At specialization time, the reduction of `(cons x y)`, where `y` is `1`, doesn't "use" the information that `y` is `1`; the question is whether `(cons x 1)` is better code than `(cons x y)`, where `y` will be bound to `1` at runtime. This situation often occurs when the usual Scheme `append` routine is specialized: each call site with an unknown first argument and a known second argument will generate a different specialization, yet all of these specializations will be identical save for a single constant in a residual application of `cons`. Is the benefit of having constants inlined worth the cost of a much larger residual program?

We are uncommitted on this issue, and believe that further research is necessary to determine when the benefits of performing a particular specialization outweigh its costs. In order to facilitate such work, FUSE is configurable: the behavior of each FUSE primitive, when faced with inputs on which it cannot be reduced at specialization time, can be individually tuned. Configuring primitives to use their arguments (*i.e.* alter their arguments' domain specifications) only when necessary to perform a reduction leads to more sharing, while configuring them to take advantage argument information for code generation leads to less sharing, but possibly faster residual programs.

In any case, limiting specialization affects only the specializer's reduce/residualize choice (*i.e.,* type 3 modification), and does not require that we alter either the re-use criterion or the computation of demand specifications.

# 4 Discussion

We have devoted much space to formalizing the problem of redundant specialization, and to describing a solution (and its variants) to the problem. However, the examples in the introduction

[14] Technically, both reduced and left residual; see [47] for an explanation.

are all very small; this begs the question of whether redundant specialization is indeed a problem in practice. We now address this issue by showing several examples where it is indeed a problem, and argue that, as specializers become more powerful, it will become more severe.

## 4.1 Examples

In this section, we describe two examples of redundant specialization which occur when specializing realistic programs with FUSE. Our first example, an interpreter for an imperative language, shows an instance of a problem which affect all specializers, while our second example shows how our re-use mechanism removes some forms of overhead which are specific to online specialization techniques.

### 4.1.1 Interpreters

Our first example is an interpreter for a small imperative language. The "MP" language, first used as an example by Sestoft [42], has `if`, `while`, and assignment statements, as well as a variety of expressions. A fragment of a direct-style interpreter for this language is given in Figure 10. The MP program shown in Figure 11 computes the $x^y$ where $x$ and $y$ are unary numbers represented as lists of arbitrary tokens.

We would expect that specializing an MP interpreter on this program would build two residual loops (*i.e.,* specializations of the procedure `mp-while`), one for the two `(while kn ...)` loops, and one for the `(while next ...)` loop. Most offline specializers (such as those described in [5, 29, 32, 42] do exactly that. They are able to build one residual loop implementing the two `(while kn ...)` loops because the interpreter procedure implementing `while` loops (`mp-while`) is invoked on identical arguments for both loops; *i.e.,* the expressions for the loop bodies are the same, the identifiers in the store are the same, and the values in the store have all been abstracted to `top` ("dynamic," in offline terminology).

Online specializers such as `FUSE`, however, do not obtain this result. Because online specializers perform abstraction only when necessary for termination purposes, they compute approximations to the values in the store. In particular, FUSE is able to deduce that the value bound to `out` is the empty list when the first `(while kn ...)` loop runs, but is a pair when the second `(while kn ...)` loop runs. Thus, the store argument to `mp-while` is different at the two call sites, causing the specializer to build two distinct specializations, one for a store with `out` known to be the empty list, and one for a store with `out` known to be a pair. Of course, since the `(while kn ...)` loops never assign or reference the variable `out`, the two specializations are identical. This wastes time in the specializer, which must construct an extra specialization, and produces a residual program containing two identical specializations.

When we enable on the re-use mechanism, things are different. After the specializer builds the first specialization for `(while kn ...)` where `out` is known to be the empty list, the domain specification of the corresponding cell in the store is `top`, indicating that the information in the call, `nil`, was unused. When asked to build a specialization where `out` is known to be pair, the specializer can deduce that it is safe to re-use the existing specialization (because (`top . top`) $\sqsubseteq$ `top`). If either of the optimality criteria of Section 3.3.1 are used, specializer will immediately deduce that the new value for `out` will not be useful, and will re-use the existing specialization, while if the simpler criterion of Section 2.3 is used, it will build a new specialization, discover that it has the same MGI as the old one, and then discard it.

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

Figure 10: Fragment of Interpreter for MP

```
(program (pars x y) (dec out next kn)
  (begin
    (:= kn y)
    (while kn
           (begin
             (:= next (cons x next))
             (:= kn (cdr kn))))
    (:= out (cons next out))
    (while next
           (if (cdr (car next))
               (begin
                 (:= next (cons (cdr (car next)) (cdr next)))
                 (while kn
                        (begin
                          (:= next (cons x next))
                          (:= kn (cdr kn))))
                 (:= out (cons next out)))
               (begin (:= next (cdr next))
                      (:= kn (cons '1 kn)))))))
```

Figure 11: Exponentiation program written in MP

This sort of difficulty is common in interpreters that maintain a store or environment representing the state of the program being interpreted. Loops in the interpreted program will force the construction of residual versions of interpreter procedures which take this state as an argument. Since most loops in a program reference only a small fraction of the variables in the state, we would expect to see many redundant specializations based on the values of unreferenced variables in the state.

However, the literature on specializing interpreters [5, 10, 17, 42, 26] fails to report such redundancies. We believe there are two reasons for this. First, most work to date on specializing interpreters has been performed with offline specializers. Such specializers treat all values which are not guaranteed to be known at specialization time as `top`; thus, the values in the state are represented either by a single `top` value (in the case of non-partially-static binding time analyses), or a `top` value per element of the state (in the case of partially-static BTA). In either case, the state is unavailable at specialization time, and cannot lead to redundant specializations. Second, most of the interpreters specialized to date have implemented small imperative languages without procedure call. When specializing an interpreter for a language without procedure call, there are few opportunities for sharing. That is, two loops in the interpreted program can be implemented by a single residual loop only when their bodies are textually identical (since their text is used in building the residual loop), which is rare. Given a language with procedures, a loop can be abstracted into a procedure and executed under multiple contexts at specialization time, leading to redundant specialization if these contexts lead to identical reductions in the body of the loop. For example, in a language with procedures, the two `(while kn ...)` loops in the program of Figure 11 would most likely be abstracted into a single procedure, which would then be invoked on two different (but, for purposes of specialization, identical) stores.

| | no reuse | reuse | improvement |
|---|---|---|---|
| specialization time (msec) | 4867 | 4816 | 1.0% |
| code generation time (msec) | 310 | 299 | 3.5% |
| residual program size (pairs) | 1180 | 1024 | 13% |
| # procedures constructed | 5 | 3 | 40% |
| # procedures in residual | 4 | 3 | 25% |

Figure 12: Results of specializing MP interpreter on exponentiation program, with and without re-use mechanism. Times are process times (not including gc time) for MIT CScheme on a 28MB NeXT workstation. The "no reuse" column is for a specializer which does not compute domain specifications at all; computing domain specifications (but not using them) takes 5658 msec.

---

One might also wonder if the need for re-use technology when specializing interpreters is limited to online specializers, since, in our example, an offline specializer's binding time analysis would simply have abstracted away the values in the state. This is certainly not the case; it is easy to envision interpreters for which offline specializers also build redundant specializations. Consider a statically typed language with parametric polymorphism. An interpreter for such a language would either factor the type descriptors into a separate type environment which is completely known at specialization time (as is done by the Algol interpreter in [13]), or would leave them attached to the values in the state (which would still allow a sufficiently precise Binding Time Analysis to deduce that they are static). In either case, the known values used by the interpreter to implement types would not be abstracted by the binding time analysis and could thus lead to redundant specialization. Offline program specializers for typed languages can treat polymorphism in programs explicitly [29], but even such specializers would need a re-use mechanism for interpreters for polymorphic languages, due to the need to represent values in the interpreted program using a single universal type.

The results of specializing the interpreter of Figure 10 on the program of Figure 11 with and without the re-use mechanism are shown in Figure 12. Re-using the same specialization of `mp-while` for both `(while kn ...)` loops yields a 13% smaller residual program with 1 less specialization. The time savings are less impressive. The specialization criterion avoided building two specializations (one is the redundant `while` loop described above; the other is an artifact of the specialization technique, and is explained in Section 4.1.2), but saved only 1% in specialization time. This is due to overhead in computing domain specifications, performing extra type comparisons, and instantiating return value templates. In this example, where the specializer spent approximately 16% of its time on such overhead, the time saved by not building the two avoided specializations is approximately equal to the overhead, yielding a better-quality residual program but no time savings. We expect to see greater time savings in the future, for several reasons. First, the redundant specialization in this example was quite small; larger programs may have more, larger, redundant specializations, giving greater savings. Second, the amount of overhead is heavily dependent on the complexity of the specializer; as we move to more complex termination mechanisms, etc, the amount of time spent computing domain specifications will become a smaller fraction of the total specialization time. Finally, the type comparison and return value instantiation operations in our prototype implementation are very slow; we expect that clever programming could speed them up by almost an order of magnitude.

### 4.1.2 Online Specialization

In the interpreter case above, the risk of redundant specialization arises from the nature of the program being specialized; the interpreter contains a procedure (`mp-while`) which, due to the structure of the MP program, is invoked on different but (for purposes of specialization) equivalent arguments at specialization time.

Sometimes, the risk of redundant specialization comes from the structure of the specializer, which specializes a particular procedure multiple times even though it is only invoked once at specialization time, and only one specialization of the procedure will appear in the residual program. In FUSE, this arises because of the fundamentally iterative nature of the specializer.

First, FUSE uses pairwise generalization [45, 40, 38] to compute what actual arguments to use when building a specialization. That is, FUSE will compute the body of a specialization over and over again on increasingly general arguments until the arguments are sufficiently general that all non-unfoldable recursive calls to the same procedure within the body of the specialization match some specialization in the cache. Thus, expressions in the body of the procedure being iteratively specialized may be evaluated on more specific values during early iterations than during the final iteration. If those expressions include procedure calls, each call site will see increasingly general arguments as iteration proceeds, meaning that any specializations built by a call site during an early iteration may be too specific for the same call site during a later iteration. Only the specializations built during the last iteration are guaranteed to appear in the residual program; the others may just be wasted effort.[15] For example, specializing

```
(define (foo x y)
  (if (= x 0)
      (bar y)
      (baz (foo (- x 1) (* x y)))))
```

on `x=top-number` and `y=1` will first specialize the body of `foo` on `x=top-number`, `y=1`, then on `x=top-number`, `y=top-number`. Thus, we will construct two specializations of `bar`, one for `y=1` and one for `y=top-number`, even though only the latter specialization will be invoked by the specialization of `foo`.

Similarly, the fixpoint techniques for computing return value approximations described in [49, 39] and Section 3.1 rebuild the body of a specialization until its return value approximation converges. If the body of a procedure $p$ invokes some other procedure $q$ on $p$'s return value, then $q$ may end up being repetitively specialized on increasingly general values. Since the only specialization of $q$ invoked by the specialization of $p$ is the one built on $p$'s final return value approximation, all but that final specialization may be useless. For example, if we assume that, in the example above, `bar` and `baz` are strict functions mapping numbers to numbers, then `baz` will be specialized first on `bottom`, then on `1`, and, finally, on `top-number`. Only this last specialization will be invoked by the residual version of `foo`.

[15] We say "may" be wasted because some other portion of the program might indeed require a specialization constructed on one of the values that was too specific for this site, in which case the work is not wasted.

Both of these situations result in the construction of potentially useless specializations. That is, if a specialization $s$ is built on argument values $v$ at a call site $c$, either (1) the arguments at $c$ in the residual program are more general than $v$, rendering $s$ inapplicable at $c$, or (2) $c$ does not appear in the residual program at all. Of course, there may be some other call site $c'$ with argument values $v'$ in the residual program at which $s$ is applicable, in which case effort was not wasted.

The re-use mechanism increases the chance that such a site $c'$ will exist by allowing $s$ to be re-used at sites where $v \sqsubseteq v' \sqsubseteq MGI[s]$[16] instead of only at sites where $v' = v$. Indeed, $s$ may be reusable at the same call site where it was constructed (*e.g.*, $c' = c$). For example, if `bar` and `baz` only use the fact that their argument is a number, then the (formerly "orphaned") specializations `(bar 1)` and `(baz 1)` can be invoked by the final specialization of `foo`. Even if this is not the case, the chance that the "orphaned" specialization will be usable somewhere else in the residual program is increased.

Such opportunities for re-use do occur in realistic programs. On such example is the second redundant specialization detected when specializing the MP interpreter (*c.f.* Figure 12. The specializer initially unfolds the invocation of `mp-while` on the `(while next ...)` loop on a store where `out` is known to be `(top . nil)`, then replaces the unfolded version with a specialization where `out` is `(top . top)`. This results in the building of two specializations of `mp-while` for the inner `(while kn ...)` loop, one for `out=(top . nil)`, and one for `out=(top . top)`, even though the first such specialization does not appear in the residual program. Thus, without the re-use mechanism, the specializer builds 5 specializations, 4 of which appear in the residual program. Since the re-use mechanism detects that the value of `out` is not used in specializing `mp-while` on either of the `(while kn ...)` loops, it builds only 3 specializations, all of which appear in the residual program.

The amount of redundancy incurred as a result of iteration in the specializer, and the amount of savings realizable from avoiding such redundancy, is highly dependent on the specializer's termination mechanism and on the complexity (height) of its type lattice. For example, under one version of FUSE with a simple type lattice and termination criterion, specializing an 800-line partial evaluator on unknown inputs built 29 specializations, 16 of which appeared in the residual program; adding the re-use criterion avoided the construction of only one specialization, and didn't even save enough time to cover the overhead of doing so, while specializing the same program under a version of FUSE with an expensive termination criterion and a larger type lattice saved a factor of 10 in specialization time.

## 4.2 Future Issues

The examples above show that our re-use mechanism can be worthwhile for current program specializers. We believe that the need for such mechanisms will only increase in the future, as specializers and the languages they operate on become more powerful.

- *Side Effects:* Existing specializers for languages with side effects make little use of information about values in the store; indeed, some, such as [6, 9] force all store operations to be residualized. Since most procedures access only a small fraction of the cells in the store, once information about values in the store is used, it will be important to base re-use decisions only on the portion of the store actually used in building the specialization (this is the same

[16] Alternately, we could use either of the more sophisticated criteria of Section 3.3.1.

as the problem of dealing with the store in an interpreter for an imperative language, except that, instead of being reified in the interpreter, the store has been moved into the language, where it must be handled explicitly by the specializer). Some work on imperative languages has encountered this problem; both [21] and [44] limit the use of the store in making re-use decisions [44].

- *Typed Languages:* The specialization of languages with types, particularly those with polymorphism and subtyping, will lead to more opportunities for redundant specialization. For example, a parametrically polymorphic procedure cannot "use" any part of its polymorphic argument, so such arguments should not be considered when making re-use decisions. Similarly, in an object-oriented language, method dispatch does not always use the exact class of an object, but merely the fact that it is a subclass of some other class, from which it inherited the method. That is, if a specialization is built on an argument of class $a$, but only invokes the argument on operations (message sends) for which $a$ doesn't override or extend the method it inherits from some superclass $b$, then the specialization is valid for all objects whose class is a subclass of $b$, not just objects of class $a$.

- *Improved BTA:* To date, redundant specialization has not been a major problem for offline specializers because the binding time analyzer limits the amount of information that the specializer considers when building specializations. As we move to more sophisticated binding time analyses, such as polyvariant BTA [32, 10, 29] and BTA which allows the use of known properties of otherwise unknown values [14], more information will be made available at specialization time. It is not clear that all of this information will indeed be useful for performing reductions; thus, we foresee an increased risk of redundant specialization.

We also foresee several uses other than the re-use of specializations for our re-use mechanism. First, with only minor modifications, it should be possible to use our mechanism to share unfolded versions of procedures as well as specialized versions of procedures. This is important when dealing with contination-passing-style code; for example, specializing

```
(lambda (k x y z)
  (if (foo x)
      (k y)
      (k z)))
```

on unknown `x` but known `k`, `y`, and `z` would normally unfold `k` on both `y` and `z`. Often, however, the residual code for both unfoldings is the same. For example, `k` might use its argument in a flow-dependent manner (*i.e.,* only use it when some free variable is true), or it might never use its argument (*i.e.,* just cons it into the final return value). Extending our mechanism to unfoldings is simply a matter of caching argument vectors and computing MGIs for unfoldings as well as for specializations. One simple way to do this is to specialize all calls, then, after specialization, post-unfold all calls with only one call site (such a 1-bit reference counting scheme is used in [6]). Of course, specializing all calls might require too much bookkeeping and caching, so some heuristic for deciding which calls are worth caching might be necessary (Similix-2 [4] chooses to specialize all `if`-expressions and closure bodies). In FUSE, post-unfolding would be quite simple; because of structure sharing in the symbolic value representation of residual code, all that is necessary is to replace the code slots of the symbolic values representing the formals with the code slots of those representing the actuals, and to re-instantiate the code for the return value.

Second, since partial evaluation can be considered to be a form of abstract interpretation [14] (though this interpretation is not necessarily over a finite-height domain), our mechanism might be useful in other abstract interpretation settings. In particular, the control-flow analyses of Shivers [43] and Harrison [24], and the type analysis of Aiken and Murphy [2, 33] must recompute the analysis of a procedure each time its abstract arguments move up in the lattice. If their abstract interpreters were to keep track of which information was actually used to perform abstract reductions during the analysis, they might be able to avoid some amount of recomputation. At this point, we cannot say what level of speedup this might provide, but since some of these analyses are quite time-consuming, re-use mechanisms might be an important part of making them practical.

# 5 Related Work

Existing program specializers use various techniques for achieving re-use of specializations. The class of *monovariant* specializers builds only one specialization of each function definition in the program, thereby achieving re-use at the expense of accuracy. The class of *polyvariant* specializers, which build a new specialization for each combination of argument values passed to a function, re-uses specializations only when their argument specifications are the same. This approach gives a more accurate specialization, but, as we have shown, can build redundant specializations.

Such behavior is less evident in polyvariant specializers used in conjunction with a monovariant binding time analysis, because the BTA limits the amount of information that the specializer considers when building specializations. Similarly, specializers with more coarse-grained argument specification models (such as those without "partially static structures" or typed unknown values) are less prone to build redundant specializations because there is less information available to cause the building of such specializations. Because this relative lack of information may adversely affect the quality of specializations, the FUSE strategy attempts to avoid placing *a priori* limitations on information use at specialization time, and instead treats the redundant specialization problem explicitly.

This section describes related work on re-use and limiting of specialization, type systems for specializers and explanation-based generalization upon which we have relied, or which is relevant to an understanding of our work and its place in the field.

## 5.1 Re-use and Limiting of Specializations

In [29][Section 7.3], Launchbury describes a specializer for a statically typed first-order language with parametric polymorphism only. Since all polymorphism is parametric, specializing with respect to polymorphic parameters would result, at best, in the inlining of constants, yielding trivial specializations. Thus, Launchbury's specializer limits itself to building specializations with respect to those portions of the available information over which the function is not polymorphic.

FUSE operates on a dynamically typed language in which both parametric and ad-hoc polymorphism (often over ad-hoc "types" built by user programs) are widely used. Since FUSE attempts to specialize functions only on information which is used to perform non-trivial reductions at specialization time, parametric polymorphism will not give rise to multiple specializations, while ad-hoc polymorphism will. Thus, we obtain the desired result without the need for assumptions about the type system.

Launchbury also suggests [29][Section 8.2.5] using projection-based strictness analysis to deduce that certain information in the arguments to a function will not be used at runtime, and having the specializer treat that function as though it had a smaller argument domain (*i.e.* without the useless information). The specializer could use this knowledge to eliminate unnecessary runtime parameter passing, and to avoid the construction of redundant specializations based on the useless information. Such a mechanism could be considered a static approximation of FUSE's; its static nature would prevent it from exploiting information that is not available until specialization time. However, from an efficiency standpoint, static analyses are attractive; we are actively investigating a combination of the two.

Gomard and Jones [21] present a specializer for an imperative language, and discuss the problem of redundant specialization due to "dead" variables assuming multiple static values. Their solution consists of a pre-processing phase which marks each program point with the names of the variables known to be live (in the sense of [1]) at that point; only the values of those variables are used in making re-use decisions. Compared with our mechanism, this scheme has two disadvantages. Because the analysis is static, it cannot handle conditional liveness based on specialization-time values. Second, since it operates at the granularity of variables rather than values, it cannot detect redundancy due to partially live structures such as the store in an interpreter.

In [15], Cooper, Hall, and Kennedy define an optimization called *procedure cloning* which has strong similarities to program specialization, and describe a re-use criterion for clones. In this approach, the programmer defines a set of targeted optimizations (*e.g.,* constant propagation) and a domain of interesting values (*e.g.,* for constant propagation, sets of <*variable name*, *constant value*> pairs). An analysis similar to specialization propagates the values through the program, computing the set of all argument vectors ("cloning vectors") for each procedure. Duplication is limited by the choice of the domain of interesting values, and by filtering the argument vectors to omit values that can't have any affect on the desired optimizations within the procedure or its callees. A second phase merges identical clones using an evaluation function that maps from argument vectors to a vector of values for "interesting" expressions which affect optimizations ("state vector"). The ability to merge clones based on state is interesting. For example, cloning can merge specializations of

```
(lambda (x y a) (* (+ x y) a))
```

for `x=1`, `y=2`, `a=top-number`, and `x=2`, `y=1`, and `a=top-number`, because `(+ x y)` evaluates to 3 in both cases. Since our algorithm views `(+ x y)` as "using" the actual values of `x` and `y`, it cannot perform this merge. This minimization is possible because only a limited subset of the expressions are "interesting;" in a program specializer where all expressions are "interesting," it would be tantamount to comparing the residual code for two specializations (*i.e.,* instead of comparing specializations on the values of their parameters, compare them on the values of all expressions reduced while computing their bodies). A third phase actually performs the cloning operation, making a copy of each procedure for each different "state vector" until a program size threshold is reached. This algorithm is potentially much more efficient than specialization because computing the "used" information for a clone requires only evaluating the "interesting" expressions, unlike specialization, which requires evaluating all (modulo control flow) expressions in a procedure's body. It is not clear how to extend this technique to program specialization.[17]

The re-use analysis described in [36] and this paper is eager, in that it treats a value as "used" (*i.e.,* reducing its domain specification) whenever a specialization-time reduction is performed which is limited to that particular value. However, not all computed values affect the building of the specialization. For example, specializing

```
(lambda (x y z)
  (let ((t (+ x y)))
    (if z
        (+ x 1)
        t)))
```

on `x=1`, `y=2`, and `z=#t` uses both `1` and `2` when computing the value of `t`, or `3`. However, since `z` is true, the computation of `t` is dead code, and does not affect the building of the specialization. Similarly, any computations performed in computing a return value are really only necessary if the specialization of the caller makes use of the return value. Katz [28] has proposed performing specialization-time reductions (and corresponding MGI calculations) lazily to avoid this problem.

### 5.2 Types

We divide existing work on type systems for specializers into two categories based on when the type analysis is performed: systems that perform type analysis at specialization time, and systems that perform it in a pre-pass, at which time only the binding times of values are known.

Several online specializers [27, 23, 3, 41, 14] maintain type information at specialization time. REDFUN-2 [23], can also propagate information out of conditionals and from the test of a conditional into its branches, but handles only scalar types (though it does compute disjoint unions, and a limited form of negation, which FUSE doesn't). In certain restricted cases, REDFUN-2 also reasons about values returned by specializations of non-recursive procedures, though it lacks a template mechanism, and thus must compute all return values explicitly. The online systems of Berlin [3] and Schooler [41] propagate information downward using *placeholders* and *partials*, respectively, both of which are similar to FUSE's symbolic values. The parameterized partial evaluation (PPE) framework of Consel and Khoo [14] is a user-extensible type system for program specialization which can infer and maintain "static information" drawn from finite semantic algebras. The online variant of PPE performs generalization to compute return values for `if` expressions, but its behavior with respect to return values of residual calls and parameters to specializations of higher-order procedures is unspecified.

The SIS[20] system uses predicates as specifications, giving finer-grained specifications than FUSE's type specifications, and offers the possibility of using theorem proving at folding time to show that re-use of specializations was proper. Unfortunately, SIS is not automatic: it lacks a theorem prover, and thus leaves all reasoning about generalization and folding to the user. Futamura's "Generalized Partial Computation"[19] also advocates the use of predicates as specifications, and

[17] If we could decide which expressions were "interesting," we could make the evaluation function explicit in the structure of the source by hoisting "interesting" expressions such as `(+ x y)` of their enclosing `lambda`; then, the parameter `3` would be identical in both cases, and reuse could take place [16]. Of course, this merely delegates the problem to each call site; taken to extremes, this would require hoisting all statically reducible expressions as high as possible.

the use of a theorem prover to further specialize the arms of conditionals based on knowledge of the test. No implementation of this scheme has, as yet, been reported in the literature.

The partially static binding time analyses of Mogensen[31] and Consel[11] reason about structured types, including recursive ones. Both operate by building finite tree representations of these data types. Consel's facet analysis [14] adds the ability to deduce that certain properties of unknown values will be known at specialization time. Launchbury's projection-based binding time analysis [29] also models recursive types; it assumes a statically typed language, and constructs a finite domain of approximations from the type declarations. These analyses only produce descriptions of structures whose size will be known at specialization time; since FUSE is an on-line specializer, it doesn't need to build recursive descriptions of such values, but instead simply operates on them. Similarly, binding time analysis can propagate information out of conditionals only when the test is static, whereas FUSE can do this in both the static and dynamic cases. Several program transformations [32, 12, 25] have been developed to address this problem of offline systems. The techniques used by partially static binding time analyses to represent specialization-time structures at BTA time may have interesting applications in online specialization. Modifying these techniques to run at specialization time (to describe runtime values),would give FUSE the ability to describe structures such as a list of unknown length that contains only integers.

Young and O'Keefe's *type evaluator* [50] is very similar to FUSE, but cannot be considered to be a program specializer because it doesn't build specializations. The type evaluator discovers types (including recursive types) using a variety of techniques, including fixpointing and generalization as used in FUSE. Unlike FUSE, however, the type analysis performed by the type evaluator is monovariant in the sense that a polymorphic formal parameter of a function will be assigned the least upper bound of the types of the corresponding actuals from all calls to the function, while a polyvariant type analysis would be free to build a separate, more accurately typed specialized version of the function for each type of actual parameter.

The FL type inferencer of Aiken and Murphy [33, 2] treats types as sets of expressions rather than sets of values, avoiding some of the difficulties usually encountered when treating function types. To some degree, FUSE uses similar techniques, specializing functions at each call site in order to compute their return types, instead of attempting to build and instantiate type signatures for functions. Our reuse mechanism adds a limited template functionality, but still stops short of computing completely general function types.

### 5.3 Explanation-Based Generalization

The process used by FUSE for computing the most general index of a specialization is isomorphic to the technique known in the machine learning community as Explanation-Based Generalization (EBG). In its usual formulation [30], EBG consists of taking an example and an explanation of the construction of the example, and performing goal regression through the explanation, producing a more general rule. In the case of a specializer, the example is a specialization, the explanation is the trace of reductions performed by the specializer while building the specialization, and the generalized result is the specialization that would be built if the same function were to be specialized on the MGI of the example specialization. We find this notion of an "explanation" unwieldy, and prefer the alternate formulation of [18], called Explanation-Based Learning (EBL), which avoids goal regression by maintaining two substitutions, SPECIFIC and GENERAL. In the case of FUSE, these substitutions correspond to the value and domain specifications of the index, respectively.

This link between specialization and EBG has been noted before by van Harmelen and Bundy [46], who showed how to convert a partial evaluator into an EBG system by leaving out unifications due to operational predicates (primitive reductions) run by the partial evaluator. However, their notion of a partial evaluator is somewhat simplistic, as it assumes that the specialization can be arrived at by merely unrolling the function description on its inputs, producing a set of leaves of the proof tree (residual applications of primitives) as a result. Their sample partial evaluator does not reason about recursion, and cannot build loops. Thus, the EBG system derived from such a simple partial evaluator can only generalize explanations that are trees, meaning it cannot generalize arbitrary recursive programs. This leaves open the possibility of performing the PE-to-EBG transformation on FUSE, possibly yielding a new class of EBG systems.

## Conclusion

We have shown that existing program specializers using polyvariant specialization can build redundant specializations. We have formulated a re-use criterion based on the domains of specializations, and have shown how an approximate version of this criterion, based on types, is implemented in our program specializer, FUSE. Adding our re-use algorithm to FUSE not only allowed it to produce residual programs with fewer specialized functions, also allowed it, in some cases, to run more quickly.

We plan to explore several avenues. We plan to add support for recursive datatypes to FUSE, increasing the accuracy of both specialization and re-use. Decreasing the granularity of the analysis could enable the specializer to re-use residual expressions, rather than just specializations, helping to build smaller residual programs. The speed of the specializer could be improved further by allowing it to re-specialize specializations instead of always specializing original function definitions (this is not possible at the moment because FUSE's input and output languages are not the same). Efficiency could also be gained by developing a version of our mechanism which works under offline specializers, which can be self-applied. More work remains to be done in the area of limiting specializations; perhaps a cost-based model, such as that used by some code generation strategies, could be used. Finally, we hope to explore the use of our specializer in explanation-based learning.


## Acknowledgements

The authors would like to thank Morry Katz for suggesting the use of types as "use" approximations, and John Lamping and Carolyn Talcott for their comments on drafts of this paper.


## References


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, 1991.

[3] A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1989. Published as Artificial Intelligence Laboratory Technical Report TR-1144.

[4] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, pages 70–87. Springer-Verlag, LNCS 432, 1990.

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

[6] A. Bondorf and O. Danvy. Automatic autoprojection for recursive equations with global variables and abstract data types. DIKU Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990.

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

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

[9] 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.

[10] 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).

[11] 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.

[12] 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.

[13] C. Consel and O. Danvy. Static and dynamic semantics processing. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 14–24. ACM, January 1991.

[14] 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.

[15] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *IEEE International Conference on Computer Languages*, Oakland, CA, April 1992. IEEE. (to appear).

[16] O. Danvy. Personal communication. March 1991.

[17] A. De Niel. Partial evaluation: Its application to compiler generator* generation. Technical Report CMU-CS-88-166, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[18] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[19] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[20] C. Ghezzi, D. Mandrioli, and A. Tecchio. Program simplification via symbolic interpretation. In S. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science. Fifth Conference, New Delhi, India. (Lecture Notes in Computer Science, Vol. 206)*, pages 116–128. Springer-Verlag, 1985.

[21] C. Gomard and N. Jones. Compiler generation by partial evaluation: A case study. *Structured Programming*, 12:123–144, 1991.

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

[23] 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.

[24] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation: An International Journal 2:3/4:*, pages 179–396, 1989.

[25] 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.

[26] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, pages 124–140. Springer-Verlag, LNCS 202, 1985.

[27] 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.

[28] M. Katz and D. Weise. Towards a new perspective on partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, San Francisco, CA, 1992. (to appear).

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

[30] T. Mitchell, M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[31] T. Mogensen. Partially static structures. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

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

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

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

[35] E. Ruf. Errata for "Using types to avoid redundant specialization". *ACM SIGPLAN Notices*, 27(1):37–39, 1992.

[36] 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.

[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. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[39] 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.

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

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

[42] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[43] O. Shivers. *Control Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. Published as technical report CMU-CS-91-145.

[44] B. Steensgaard. Personal communication. October 1991.

[45] 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.

[46] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36(3):401–412, October 1988.

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

[48] 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.

[49] 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.

[50] J. Young and P. O'Keefe. Experience with a type evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581. North-Holland, 1988.