Using Types to Avoid Redundant Specialization[*] FUSE MEMO 91-5

Erik Ruf[†]
Daniel Weise[‡]
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-2140

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, and describe an approximate implementation of this criterion based on types, and its implementation in our partial evaluation system, FUSE. Finally, we relate our algorithm to existing work in partial evaluation and machine learning.

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*[3] operations. Folding is done by recursively specializing certain distinguished[1] parts of the program (*specialization points*), and re-using these specializations where appropriate. Most specializers use a strategy called *polyvariant specialization*[2], 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 [8, 17, 21] 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[16]; thus, both programs and specialization points will be user function definitions. This is true of most specializers for functional languages, although some, like Similix[1], 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

[*]Initially published in *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Directed Program Manipulation*, ACM Press, pp. 321-333, June, 1991. This version corrects a couple small errors in Section 2.3 of the published paper.

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

[‡]Supported in part by NSF Contract No. MIP-8902764, and in part by Defense Advanced Research Projects Agency Contract No. N00014-87-K-0828.

[1]Many program specializers, such as Mix[9] 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[20] and Turchin's supercompiler[18].

than "static" and "dynamic," because these latter terms may connote approximations; in systems using binding time analysis[9], "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]

```
(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 or not 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, as in

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

which, if **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 or not **(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)))))
```

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))
```

This specialization 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 specified by the known argument values 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 four 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 modifications to the re-use mechanism to extend its functionality (Section 3), and a discussion of related work (Section 4).

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

[2]We realize that, under most specializers, this function (and many of our other examples) would never be specialized, but only unfolded. However, by introducing recursion, we can trivially convert all of our examples to ones which must be specialized in order to guarantee termination of the program specializer. For the case of the test1 function in this example, consider instead specializing

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

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

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

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 necessary for the specializer to 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 the specialized (residual) function r, f, and s, 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 shown in the Introduction, 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. The reason is that 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*

$$DOS[r] = \{v \in V \mid (E[f]v = E[r]v) \vee (E[f]v = \perp_V)\}.$$

The domain of specialization is the set of argument values under 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 $C[s] \subseteq DOS[r]$.*

Unfortunately, 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)`. Knowing the domain of a specialization doesn't tell us whether or not 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`, `(lambda (x y) 3)` is strictly more specialized than `(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:

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

1. The real specializer is a program, not a mathematical function. It operates solely on representations of values (specifications) and never on the values themselves. Thus, it cannot compare two specifications extensionally, or compute the domain of specification, since these require executing the concretization function C, which is not enumerable. Instead, it can only compare specifications directly, and must compute a specification of the DOS instead of the DOS itself. Unless specifications can denote arbitrary, possibly infinite sets of values (such as the set of pairs of numbers whose sum is three), we will have to settle for approximations. Our strategy will be to choose a representation that provides acceptable approximations while remaining simple to compute.

2. The optimality test requires that the specializer compute the specialization $SP(f, s)$ before deciding whether or not 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.

2 Re-use of Specializations in FUSE

The remainder of this paper describes details of FUSE[20], 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 [16] 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. Obvious choices for this purpose are the top elements of the various subdomains in VAL, since each top element can be interpreted as describing all of the elements below it (this corresponds to the notion of an ideal in the value domain; for details, see [12]). Therefore, in addition to values, FUSE 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 $< \top_{VAL}, \top_{VAL} >$.

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

In actuality, the FUSE type system is slightly more complicated. In order to support fixpointing, a representation for $\bot_{VAL}$ 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 [21].

2.2 Computing Approximations of the DOS

Now that we have defined 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 complete, 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. 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. 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 set 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 set 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.

- *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, while if it is a user procedure, the specializer must make the usual unfold/specialize decision. If the decision is to specialize, and a previous specialization is re-used (how this choice is made is described in the next section), then the domain specification of each actual argument must be set to the domain specification of the corresponding argument in the MGI of the re-used specialization.

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. For an example, see Figure 3.

2.3 Using the Approximations to Control Re-Use

The specializer computes an approximation to the domain of specialization of each specialization it builds. This approximation, the MGI, is cached along with the specialization index and the specialized function body. As stated above, having the MGI allows us to decide when it is safe to re-use 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 MGI's, 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.

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

A 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 symbolic value a having value specification v, producing specialization r having domain specification (MGI) d. When specializing f on symbolic value b having value specification v' where $v \sqsubseteq v' \sqsubseteq d$, specializing f on b 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 for b.
2. $v \sqsubseteq v' \sqsubseteq d$. In this case, b 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, b denotes a set of values which is not a superset of those denoted by a. Even though it is safe to re-use r, building a specialization for b might be worthwhile. Therefore, build a new specialization n; if it is more specialized than r (by the criterion of Definition 4), then use it, otherwise discard it and re-use r. In the latter case, discard only the body of the new specialization n; a relation between the index of n (namely, v') and the MGI and body of the old specialization r is still cached. Thus the knowledge that r can be re-used for specification v' is not lost, saving the specializer from performing this needless computation again.

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 on whether or not 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. 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

```
;;; 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-num})
  (value ())}
```

Figure 2: Examples of value specifications

```
;;; 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 typechecks
```

Figure 3: Examples of specializations and their MGI's

```
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 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 4: FUSE's algorithm for re-using specializations

specializations of f are complete.

FUSE's algorithm for re-using specializations is shown in Figure 4. As an example, consider the function definition

```
(define t1
  '(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)))))))
       (cons (f x z)
             (f y z))))).
```

Using

```
(pp-graph (sp t1 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

```
(pp-graph (sp t1 (a-number) 1 (a-value)))
```

to invoke the specializer produces a different behavior. In this case, the first specialization built is for `n` specified as `top-number`. This type information is used, and thus the MGI also specifies `n` as `top-number`. When the second call to `f` is specialized, the value of `n`, namely `(number 1)`, is more specific than the MGI 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. However, the specializer does learn from building and discarding the second specialization: it now knows that the first specialization is applicable to all known numbers (because there is no number with a value specification v such that $v \sqsubset$ `(number 1)`). So if, at this point, the program were to call `f` on `n=2`, the specializer would be able to re-use the specialization `f6.1` without further ado.

Note that if, in addition to computing the domain specifications (*i.e.*, which information was actually used), FUSE were to also compute which information was demanded but unavailable, this situation would not have arisen. Instead, after computing the first specialization (with `n` as `top-number`), the specializer would have known that the numerical value of `n` was not demanded, and would not have built and discarded the specialization for `n` as `1`. This enhancement will be implemented in a future version of FUSE.

3 Modifications

This section describes three modifications to the re-use algorithm given above. The first two modifications are necessary because FUSE, unlike previous specializers, reasons about the values returned by residual conditional expressions and calls to specialized procedures. The third modification extends the usefulness of our method by allowing FUSE to limit the use of information in creating specializations, in order to avoid creating large numbers of "trivial" specializations that provide only minimal improvements at runtime.

3.1 Residual Conditionals

Most specializers are unable to export information out of an `if`-expression with unknown test. FUSE, however, is often able to compute useful information about such constructs, by returning the generalization of the value specifications of the two arms of the conditional. For instance, specializing

```
(lambda (p x y z)
  (+ 1
     (cdr
       (if p
           (cons x z)
           (cons y z)))))
```

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

```
(lambda (p x y z)
  (+ 1 z))
```

under FUSE. 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)
  (+ 1
     (if p a b)))
```

where `p` is unknown, and `a` and `b` are both known to be `2`. The result of `(if p a b)` will be a symbolic value with value specification `(number 2)` and domain specification `top`. Applying the primitive `+` will make use of the fact that its input is `2`, and will change this domain specification to `(number 2)`. Unfortunately, the domain specifications for `a` and `b` were never changed; therefore, the MGI for the the specialization will be `(top top top)` when it should be `(top (number 2) (number 2))`.

The solution is fairly obvious: whenever the domain specification of a symbolic value obtained from a generalization is changed, the domain specifications of both inputs to the generalization must also be changed accordingly. Therefore, symbolic values contain a `parents` attribute; 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 t a b)` returns a symbolic value s with value specification `(number 2)`, domain specification `top`, and parents `(a b)`. When the `+` operator sets the domain specification of its input, s, to `(number 2)`, it also sets the domain specifications of s's parents, `a` and `b`, to `(number 2)`, and the correct MGI is computed.

3.2 Residual Function Calls

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 [21]).[5] 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.

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-num`, and `z` specified as `top` yields

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

[5]The utility of this feature will increase significantly once FUSE has the ability to describe recursive data types. For now, return values of unknown size can only be described by `top` or `top-list`, neither of which gives any information about the components of such values.

a return value specification of `top-num`, 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-num`. 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. The latter is more attractive, as it preserves more information, but it would seem to require that recomputing the body of the specialization on each set of actual parameter specifications. As it turns out, we need not perform such a recomputation. 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, or
2. computed by the procedure, or
3. all or part of one of the arguments to the procedure, returned unaltered.

These are the only possibilities. Constants are not a problem, as they will be the same for every call site. Items computed inside 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 compute the same values will re-use this specialization. This leaves the case of arguments that are "passed through," which is easily 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 actual argument specifications against those of 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.

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 significantly reduce specialization time without compromising the accuracy of specialization.

3.3 Limiting Specialization

The re-use analysis described above allows for the re-use of a specialization in cases where the arguments are more general than the arguments on which the specialization was originally computed, in the case when not all of the information in the original argument (specialization index) is 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 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 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[6] 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 `x` 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

[6] Technically, both reduced and left residual; see [20] for an explanation.

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.

4 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.

4.1 Re-use and Limiting of Specializations

In [11][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 [11][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.

4.2 Types

Existing work on type systems for specializers can be divided into two categories based on when the type analysis is performed: those that perform type analysis at specialization time, and those that perform it in a prepass, at which time only the binding times of values are known.

The specializers written by Kahn [10], Haraldsson [8], and Schooler [17] maintain type information at specialization time. Haraldsson's system, REDFUN-2, 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 type system of Schooler's IBL is very similar to FUSE's type system: IBL uses *partial values* that correspond closely to FUSE's symbolic values.

The SIS[7] 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"[6] also advocates the use of predicates as specifications, and the use of a theorem prover to further specialize the arms of conditionals based on knowledge of the test.

The partially static binding time analyses of Mogensen[14] and Consel[4] reason about structured types, including recursive ones. Both operate by building finite tree representations of these data types. Launchbury's projection-based binding time analysis [11] 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. In [15][Section 9.4], Mogensen presents a partial solution to this limitation, based on the use of a rewrite rule; unfortunately, this technique can result in code duplication. We find the partially static BTA techniques interesting, and intend to investigate modifying them to run at specialization time, which would give FUSE the ability to describe structures such as a list of unknown length that contains only integers.

Young and O'Keefe wrote a *type evaluator*[22], which 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.

4.3 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 [13], 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 [5], 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 [19], 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. This speedup effect can be large; for example, when specializing a large program (a partial evaluator for Scheme, comprising approximately 800 lines of code) on completely unknown inputs, our re-use algorithm was able to avoid building 115 specializations, and discarded another 3 after they were built, saving a factor of 10 in specialization time.

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). 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.

Acknowledgement

The authors would like to thank Morry Katz for offering the insight that the amount of information about a value "used" in a specialization can be concisely approximated using the type lattice, and for many discussions on this and similar issues.

References

[1] 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.

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, University of Glasgow, 1991. Published as CS Report CSC 90/R2.

[12] D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

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

[14] 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.

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

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

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

[18] V. Turchin. The algorithmn 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.

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

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

[21] D. Weise and E. Ruf. Computing types during program specialization. Technical Report CSL-TR-90-441, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990.

[22] 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.