Computing Types During Partial Evaluation

Daniel Weise and Erik Ruf

Fuse Memo 90-3-revised
December 1990

Computer Systems Laboratory
Department of Electrical Engineering
Margaret Jacks Hall Room 426
Stanford University
Stanford, California 94305-2140

Abstract: We have developed techniques for obtaining and using type information during program specialization (partial evaluation). Computed along with every residual expression and every specialized program is type information that bounds the possible values that the specialized program will compute at run time. The three keystones of this research are *symbolic values* that represent both the set of values that might be computed at runtime and the code for creating the runtime value, *generalization of symbolic values*, and the use of *online fixed-point iterations* for computing the type of values returned by specialized recursive functions. This work differs from previous specializers in computing type information for *all* residual expressions, including residual `if` expressions, and residual calls to specialized user functions. The specializer exploits the type information it computes to increase the efficiency of specialized functions. In particular, this research allows the class of recursive descent programs that use data structures to be more efficiently specialized.

Introduction

A *program specializer* (also called a *partial evaluator*) transforms a program and a description of the possible inputs of the program into a *specialized program* that is optimized for those inputs. For example, a three argument function could be specialized for its first argument being 2, its second argument being any value at all, and its third argument being an integer. Program specialization is an experimentally proven technique for creating compilers and compiler generators [11], automatically rederiving important algorithms [7], speeding up computations by two orders of magnitude [5], parallelizing scientific code [3], and optimizing programs [4].

Program specializers operate by symbolically executing the program on the inputs. Expressions that can be reduced are reduced, while those that cannot are left *residual* in the specialized program. A major goal of specialization is accuracy: getting as close to the most specialized program as possible. The accuracy of program specialization, and the speed of specialized programs, are both limited by the amount of available information used by the specializer. Program specializers often ignore or throw away information to either simplify their task or to achieve termination. Obviously, the more accurate a program specializer is, the more useful it is, and the larger class of programs that will benefit from program specialization.

This paper addresses the importance of type information to increase the accuracy of specialization. Type information bounds the possible values that will be returned by functions and expressions at runtime. (That is, when the specialized program executes.) Our goal is to compute type information for all residual expressions, especially residual conditionals and residual function calls. The previous specializers that have computed type information [8, 17] did not compute the types of specialized subprograms, and [8] did not handle recursive types (for example, it was unable to represent a list three elements long whose first element was the symbol `a`). Specializers that do not compute complete type information lose accuracy, and therefore produce slower specialized programs. Programs can be rewritten (restaged, in program transformation parlance [12]) in a continuation passing style to avoid losing type information, and therefore avoid losing accuracy. Unfortunately, this can interfere with extant unfolding strategies, and can result in very poor specialization. Our methods allow for accurate partial evaluation of naturally written programs, without interfering with the unfolding strategies that decide whether to unfold or specialize a given function call.

The paper considers the specialization of programs written in a Scheme-like eager untyped functional programming language that operates over integers, booleans, procedures, and pairs. For simplicity of exposition, we assume that the specialized program is first order, though this is not a requirement upon the source program. Although this paper focuses on an untyped language, it should become clear that our methods are also important for typed languages, such as ML. Many programs are written using inductive types (e.g., lists) because they must operate on inputs of unknown size. However, once the part of the input is supplied, the specialized program could use tuples, which are more efficient than lists, or could at least compile away much of the runtime type checking and dispatching on union types. Our system compiles away this extraneous type checking and dispatching that even typed languages such as ML must perform at runtime.

As an example of the kind of type information we wish to compute, consider a store represented as an association list that binds names to values (Figure 1). The store is updated without the use of side effects, and without changing its shape. The recursive function `process-updates` accepts a list of updates and a store, and returns the store resulting from processing the updates. Our methods will prove, for example, that the specialization of `update-store` on an unknown `name` and `value`, and a store of the form `((a . <some-value>) (b . <some-value>) (c .  <some-value>))` will return a store of the same form. Doing so requires reasoning about recursive types and residual `if`-expressions. Similarly, our methods will prove that the specialization of `process-updates` on an unknown set of updates and a store of the form `((a . <some-value>) (b . <some-value>) (c . <some-value>))` will return a store of the same form. Doing so requires performing a fixed-point analysis. The appendices present the operation of our program specializer on an interpreter for a very simple language.

As another, more important and realistic example, consider a fragment of an interpreter for the MP language[1], whose syntax is

```
Program ::= (program (pars Id*) (dec Id*) Command)
Command ::= (:= Id Exp) |
```

[1] The appendices contain more information about the MP language.

```
(define (update-store store name new-value)
  (cond ((null? store) '())
        ((eq? (caar store) name)
         (cons (cons (caar store) new-value) (cdr store)))
        (else (cons (car store) (update-store (cdr store) name new-value)))))

(define (process-updates updates store)
  (if (null? updates)
      store
      (process-updates
        (cdr updates)
        (update-store store (caar updates) (cdar updates) store))))

(define (lookup-in-store name store)
  (if (eq? name (caar store))
      (cdar store)
      (lookup-in-store name (cdr store))))
```

Figure 1: Functionally Updating a Store. Unless information about the return values of residual `if`-expressions and residual recursive functions is computed, a program specializer will lose vital information when specializing either function for a partially or fully known store when the other inputs are unknown.

```
             (if Exp1 Exp2 Exp3) |
             (while Exp Command) |
             (begin Command*)
Exp     ::= <not provided>
```

Commands map an initial store into a final store. The interpreter starts by building a store that binds each name declared in the `pars` section to the corresponding input, and each name declared in the `dec` section to the empty list. During interpretation the structure of the store doesn't change.

The fragment of a standard recursive descent interpreter, called `mp-interp`, that interprets commands appears in Figure 2. Unless type information is computed for residual if and `call` expressions, compiling a MP program by specializing `mp-interp` on an MP program will yield a poor specialization because knowledge of the structure of the store will be lost during compilation. Residual Lookups and updates to the store will be runtime loops instead of dead-reckoned car and cdr operations.

For example, consider the MP program fragment

```
(begin (while (not (= z 0)) (begin (:= z (- z 1))
                                   (:= a (cons (quote 1) a))))
       (if (= x y) (:= x 1) (:= y 1)))
```

Assuming that the structure of the store is known to be, say `((x . ?) (y . ?) (z . ?) (a . ?))`, before the compilation of this statement, when the `while` portion is compiled the structure of the store will be lost if type information is not computed for residual specialized functions. This is because the specialization (compilation) of the `while` statement is a residual recursive procedure. When the `if` statement is compiled, lookups and updates to the store will not be compiled into straight-line code. Instead, there will be runtime loops. A similar problem occurs when compiling `if` statements and the specializer does not compute the types of residual `if expressions`: even when the structure of the store is known before compiling an `if` statement, it will be lost after compiling the `if` statement.

We find this state of events unacceptable. It reduces the applicability and power of program specialization. The workaround to this problem is to rewrite programs so that they don't return structured values, instead only passing them downwards. For example, the MP interpreter presented in [14] is written in a quasi-CPS style (the rest of the computation is encoded as an MP program) to ensure that the store is only ever passed

```
(define (mp-interp program input)
    (letrec
      ...
      (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)
             (let ((test-exp (cadr com)) (while-com (caddr com)))
               (let while-loop ((store store))
                 (if (mp-exp text-exp store)
                     (while-loop (mp-command while-com store))
                     store))))
            ((eq? token 'begin)
             (let begin-loop ((coms rest) (store store))
               (if (null? coms)
                   store
                   (begin-loop (cdr coms) (mp-command (car coms) store)))))
            ....
```

Figure 2: The fragment of the MP language interpreter that interprets commands. The interpreter in written in a standard recursive descent style.

downward.[2] We believe it is preferable to extend the power of program specialization to avoid the necessity of such rewrites. This research does just that.

The three key mechanisms used to compute type information are: *symbolic values* that represent a runtime value, and which consist of a *type object* that denotes of the set possible runtime values, and a code expression indicating how to compute the value; *generalizing two symbolic values* to create a new symbolic value representing the values of each of them; and the use of *online fixed-point computations* for computing the return type of specialized recursive closures. Symbolic values and generalization have been presented before to remedy other problems in online program specialization such as code duplication, type propagation, overly strict interpretation of values versus code, and termination [20] ([5] uses *placeholders*, which are very similar to symbolic values). The technical advance of this research over [20] is a different treatment of `if` expressions and performing fixed-point iterations when specializing closures.

This research allows for *abstract interpretation with concrete values* (AICV). This is a method of performing abstract interpretation with concrete values where possible. Whereas all values must be abstracted prior to standard abstract interpretation, our specializer only abstracts values as needed to achieve termination. This strategy allows data-dependencies to be considered during the interpretation. For example, when the specialized program is first order, our specializer can be used without change to perform strictness analysis. Only a minor change is needed to run the standard example of computing whether a program manipulates 0, positive numbers, or negative numbers. And because concrete values are allowed, where it can be shown the result is a concrete number (say, as part of a list), that will be done. Performing strictness analysis for given arguments being positive or negative numbers is another possibility. The termination of AICV is directly related to that of the program specializer itself.

Related Research

Binding Time Analysis (BTA), an important tool for constructing self-applicable specializers [11, 15, 6, 10], is related to our research. BTA uses abstract interpretation to prove global properties of recursive programs to be specialized. These properties are exploited during specialization. BTA is an offline analysis; it proceeds with no knowledge of the data values the program will be specialized on. It derives what will and will not be known during specialization, and, in the case of *partially static structures* [14, 6], the shape of the data. For example, on the store example above, BTA will determine that the return result of `process-updates` is a list of pairs whose cars are known and whose cdrs are unknown. However, because BTA is deprived of actual data values, it is not as precise as our method, which does have access to actual data values. Computing the return types of residual `if` expressions and residual recursive functions requires online generalization and online fixed-point computations, respectively. Offline specializers are designed specifically not to perform these computations at runtime.

This research is closely related to that of Young and O'Keefe [21], who created what they called a *type evaluator* for an untyped functional, lazy dialect of Scheme. They created a system that shared many features of a program specializer. The goal of their system was to determine, as accurately as possible, the possible values that a function would return. Their type system differs from ours in two ways. First, we operate in an applicative dialect of Scheme with finite types, while they operated on a lazy language dialect of Scheme with infinite (circular) types. Second, we model nearly the exact structure of the standard value domain, whereas their model of the standard value domain is richer than ours, and carries more information. For example, in our system, if one arm of an if-expression returns a number, and the other arm returns a pair, we declare that the expression returns $\top$, which is the least upper bound of these values in the standard value domain. Their system is capable of remembering that the value is either a pair or a number.

Our research effectively weds their type evaluator to a program specializer. The specializer maps a program and the set of values the program will be applied to (the program's domain) into a specialized program and the range of the specialized program. This marriage helps both the specializer, which becomes more accurate, and the type evaluator, which also becomes more accurate. The same mechanisms that cause the specializer to terminate cause the "type evaluator" to terminate. And, of course, the type information

[2] It is not always possible to perform the rewrite and retain good specialization. For example, although Mogensen's MP intepreter does not lose the structure of the store when is it used to compile programs, is written in a style that confounds virtually all automatic termination strategies. Mogensen manually places unfolding annotations on his interpreter after his binding time analysis runs.

is synergistically used in the termination strategies.

Haraldsson's REDFUN-2 [8], a redesign of [2], was designed to compute and track type information during specialization. For each expression REDFUN would compute either a set of concrete values that the expression could return at runtime, a set of concrete values that the expression could not return at runtime, or a scalar type descriptor, such as `SEXPR`, `INTEGER`, or `STRING`. REDFUN-2 goes further than our system in maintaining sets of concrete values, but its datatype descriptors, being only single markers, are weaker than ours, which are recursive. In particular, our system is powerful enough to solve problems Haraldsson cites as future research. Another important difference is that REDFUN-2 only automatically gets type information from residual function calls in very restricted situations. Also, it is not automatic, as it relies on programmer advice to carry out its operations.

Also of extreme relevance is the work of Schooler [17], who also propagated type information. His *partials* are similar in spirit and structure to our symbolic values. The overlap in method is closest in the handling of structured values and known values. However, there are several fundamental differences. First, no automatically derived type information propagates out of `if`-expressions or residual function calls. Second, partials are used for strict substitution of code, which results in common subexpression introduction (code duplication), a serious problem. Third, because of the textually substitutive use of partials, his system resorts to explicit environment manipulations to properly resolve variable lookups. Our system, which resolves textual issues in a separate pass, moves the complexity out of the specializer [19]. Finally, his system is not automatic. Some of the complexity of his system is due to delaying side-effects until runtime, which the system described in this paper does not have to face.

This paper has 6 sections. We start by defining and discussing program specialization, and type approximations. The second section defines the syntax and standard semantics of the language whose programs will be specialized. The third section discusses *symbolic values*, the result of evaluating an expression is always a symbolic value. The specializer and generalization are presented in the fourth section. Fixed-point iterations are presented in Section 5. Finally, Section 6 discusses concrete abstract interpretation.

1 Program Specialization

The user of a specializer specifies the set of values the program is to be specialized for. An element of this set is called a *valid input*. The set can be as small as a single value, such as the number 5 or the pair `(a . b)`; an entire class, such as the integers; structured values that themselves contain classes, such as a pair that contains 5 and the class of integers; or the entire set of denotable values, which we write as `a-value`, and which others write as `U`. Different specializers allow different classes to be described, but they all share the same approach: the domain[3] the program is to be specialized for is provided, and the specializer creates a program specialized for that domain. We use symbolic values (Section 3) for expressing domains and ranges.

We define a program specializer as follows.

Definition 1 *(A Strict Program Specializer) Let* f *be a program in $\mathcal{L}$, V be the value domain of $\mathcal{L}$, sarg $\in$ Sarg be the representation of a consistent subset of V, $T[]$ be a function mapping Sarg into the subset it denotes, and let $E[]$ map programs to their meanings. A* strict program specializer *is a program SP mapping $\mathcal{L} \times$ Sarg into $\mathcal{L}$ such that*

$$\forall a \in T[sarg](E[f]a = E[SP(f, sarg)]a)$$

whenever SP(f,sarg) is defined (terminates).

Definition 2 *(A Lazy Program Specializer) Assuming the conditions of the previous definition, a* lazy program specializer *is a program SP such that*

$$\forall a \in T[sarg](E[f]a \sqsubseteq E[SP(f, sarg)]a)$$

The only difference between the two definitions is the use of $=$ versus $\sqsubseteq$ in the consequents. Strict specializers produce specialized programs that terminate exactly when the original program would terminate.

[3]This use of the term "domain" is with respect to the domain/range of a function, not semantic domains.

Lazy specializers produce specialized programs that terminate more often than the original program does. That is, they introduce lazyness into the program; computations that would cause divergence are either excised because they are dead code, or they are moved from applicative positions (arguments to functions) to non-applicative positions (arms of conditionals). We will not rehearse the pros and cons of strict versus lazy program specializers. The specializer described in this paper is lazy. Our definition of program specialization subsumes that of Jones [9], who describes a specializer as a program that accepts *part of* a program's inputs, and that of Launchbury [13] who describes partial evaluation in terms of projections. Their definitions also stipulate that the specialized program only require the "part of the input" that wasn't provided at partial evaluation time.

Our prior definition of program specialization assumed that the specialized program consisted solely of code, *i.e.*, was an expression. In this research, specialization produces both a code attribute and a type attribute, so we need to extend the definition of program specialization. A *typing program specializer* (TPS) produces specialized programs that have a code attribute and a type attribute. Let the function SF_{code} return a specialized program's code attribute, and the function SF_{type} return its type attribute. Informally, a typing partial evaluator accepts a program and the domain the program will be applied to, and returns both a specialized program and its range.

Definition 3 *(A Lazy Typing Program Specializer) Assuming the conditions of Definition 1, a* lazy typing program specializer *is a program TSP mapping $\mathcal{L} \times Sarg$ into $\mathcal{L} \times Sarg$ such that*

$$\forall a \in T[sarg](E[f]a \sqsubseteq E[SF_{code}(TSP(f, sarg))]a)$$

A *strict typing program specializer* is similarly defined, substituting $=$ for $\sqsubseteq$.

We define the *safety property* of a typing program specializer similarly to the notion of safety in abstract interpretation [1] and to the definition of [21]:

Definition 4 *(Safety of Typing Program Specialization) Assuming the conditions of Definition 3, a typing program specializer TSP is* safe *if*

$$\forall a \in T[sarg](E[f]a \in T[SF_{type}(TSP(f, sarg))])$$

This definition states that the value produced by applying the original function to any value input is within the type computed by the specializer. We state without proof that our specializer is safe.

2 Syntax and Value Domain

This paper considers a simple untyped eager functional programming language operating over numbers, booleans, pairs, symbols, and `nil`. The abstract syntax of the language is:

`i`	$\in$	*Id*	Identifiers
`e, body`	$\in$	*Exp*	Expressions
`n`	$\in$	*Num*	Numerals
`b`	$\in$	`t | f`	Boolean constants
`e`	`::=`	`i | n | b |`	
		`(if e1 e2 e3) |`	Two armed conditional
		`(lambda i* body) |`	Lambda Abstraction
		`(call e1 e*) |`	Function Application
		`(letrec (i e)* body)`	Mutually Recursive Definitions

The initial environment provides *nil* and all the primitive functions. Section 3 presents the implementation of the primitive functions `+`, `cons`, and `car`. All other primitives are a variation of these three.

The value domains are

$$\begin{aligned}
XNum &= Num + \texttt{a-natural} \\
XBool &= \texttt{t} + \texttt{f} + \texttt{a-boolean} \\
XPair &= SVAL \times SVAL \\
Nil &= \texttt{nil} \\
XFunc &= exp \times Env + \texttt{a-function} \\
XVAL &= XNum + XBool + XPair + Nil + XFunc + \\
&\quad \texttt{a-value} + \texttt{no-value} + \texttt{a-list} \\
\\
sval \in SVAL &= XVAL \times Code \\
\\
Code &= Id && \text{Formal Parameter} \\
&\quad + Id \times Code^* && \text{Residual Function Call} \\
&\quad + Code \times Code \times Code && \text{If expression} \\
&\quad + XVAL && \text{Known Value} \\
Env &= (Id \times SVAL)^* \\
SC &= Id \times Index \times SVAL && \text{Specialized Closure} \\
Index &= SVAL^* \\
cache \in Cache &= XFunc \times Index \rightarrow (SF + \texttt{empty})
\end{aligned}$$

Figure 3: Domains used by the program specializer when interpreting programs. The specializer only manipulates the representation of objects, not the objects themselves. For example, numbers are represented by *Num*, rather than *Nat*. A symbolic value (*SVAL*) contains an element of *XVAL* and an element of *Code*. *Code* is either the name of a primitive, the name of a specialized closure and arguments, or an *XVAL*. (The implementation differs from this description on the structure of *Code* elements. In the implementation the subfields are all *SVAL*s rather than *Code*, but it is easier to describe the system using just *Code* elements. The implementation keeps *SVAL*s for the type information they encode; the code generator uses this information.) *SC* stands for Specialized Closure; it consists of a name, the arguments the original closure was specialized on, and the symbolic value representing both the body of the specialized closure and its return type. The cache maps a closure C and a list of symbolic value into the specialized closure that C was specialized into, if any. The marker `a-list` represents the generalization of `nil` and a pair.

$$\begin{aligned}
Nat &= \text{the flat domain of natural numbers} \\
Bool &= true + false \\
Pair &= VAL \times VAL \\
Nil &= nil \\
Func &= VAL \rightarrow VAL \\
VAL &= Nat + Bool + Pair + Nil + Func
\end{aligned}$$

3 Symbolic Values

Symbolic values represent both the object that will be produced at run time and the code to produce the value (Figure 3). The result of partially evaluating any expression is a symbolic value. Each symbolic value has a *value attribute* and a *code attribute*. The value attribute represents a subset of *VAL*. We say that a symbolic value represents a *known value*, or, more simply, *is* a known value, when the set it represents contains only one value. Any symbolic value whose value attribute is **`a-value`** is called *completely unknown* or *completely unconstrained*. Besides representing a subset of V, the value attribute is also a value that is operated upon. For example, during partial evaluation the **`car`** operation takes the car of value attribute of the symbolic value it is applied to. We will notate a symbolic value using angle brackets, for example, a symbolic value with value attribute v and code attribute c will be written $\langle$v,c$\rangle$.

The code attribute specifies how the value represented by the symbolic value is computed from the formal parameters of the procedure (or, when block structure is employed, procedures) being specialized. The code specifying the creation of a value V only appears in the specialized program when V itself must be constructed at runtime. For example, consider a cons-cell that is constructed at partial evaluation time but is not needed at runtime. The instructions for constructing the cons-cell will not appear in the specialized program. When a symbolic value represents a known value, the code attribute is the value attribute. The code is expressed as a graph [19], so a code generator is used to express the code in some particular programming language.

Example symbolic values include:

⟨4,4⟩	The number 4
⟨`a-natural`,`a`⟩	A number whose name is `a`
⟨(⟨4,4⟩ . ⟨`a-natural`,`a`⟩),`(cons 4 a)`⟩	The cons of the above 2 items.
⟨(⟨`a-boolean`,`(car (f a))`⟩ . ⟨`a-natural`,`(cdr (f a))`⟩),`(f a)`⟩	A pair that will be created by a runtime call to `f`.

Even though it appears as if there is code duplication (for example, `(f a)` appears twice), this is only an artifact of presenting a graph without arrow to show sharing.

The subset of VAL represented by an $XVAL$, and by extension, a symbolic value, is given by the function T:

$$\begin{aligned}
\mathrm{T}[\texttt{a-natural}] &= \{0, 1, 2, \ldots\} \\
\mathrm{T}[\texttt{Num}] &= \{\mathrm{E}[\texttt{Num}]\} \\
\mathrm{T}[\texttt{a-boolean}] &= \{true, false\} \\
\mathrm{T}[\texttt{t}] &= \{true\} \\
\mathrm{T}[\texttt{f}] &= \{false\} \\
\mathrm{T}[\texttt{a-function}] &= Func \\
\mathrm{T}[\texttt{a-value}] &= \text{Concrete elements of } Val \\
\mathrm{T}[\texttt{no-value}] &= \{\} \\
\mathrm{T}[\langle v_1, c_1\rangle \times \langle v_2, c_2\rangle] &= \{a \times b | a \in T[\mathbf{v_1}], b \in T[\mathbf{v_2}]\}
\end{aligned}$$

We define T applied to a symbolic value to be T applied to the symbolic value's value attribute.

There is a parallel between `a-natural`, `a-boolean`, `no-value`, and `a-value`, and the domain values $\top_{nat}$, $\top_{bool}$, $\bot_{VAL}$, and $\top_{VAL}$, respectively. In each case, the marker represents the downward closure (minus inconsistent elements that contain top or bottom elements) of the corresponding top element.

The specializer is invoked on a program and a symbolic value for each argument. The user constructs symbolic values using the unary function `known-value`, which is applied to numbers and booleans, the niladic functions `a-natural`, `a-value`, `a-boolean`, and the binary function `a-cons`.[4] The symbolic values so constructed have empty code slots.

The specializer *instantiates* the symbolic values before starting the symbolic execution. Instantiation accepts a symbolic value and code, and makes the code slot of the symbolic value be the code. For pairs, it also calls itself recursively, passing itself the car of the pair and code representing the `car` of the parent, and the cdr of the pair and code representing the `cdr` of the parent.[5] When instantiating user provided symbolic values, the code is the formal parameter each symbolic value is bound to. Instantiation is also invoked when computing the return value of residual function calls: the symbolic value representing the body of the called function is instantiated with code representing the call.

For example, the user can describe a data structure of the form `((a . ?) (b . ?))` via `(a-list (a-cons 'a (a-value)) (a-cons 'b (a-value)))`. This creates a symbolic value of the form (unfilled code slots):

[4] No function is provided building the equivalent of `no-value`. This value is used by the specializer as the initial approximation when computing fixed-points. If we were to specialize programs written in a lazy language, it would be worthwhile to allow the user to specify `no-value` for performing strictness analysis (*cf* Section 6).

[5] Actually, the code slot is set to take the `tc-car` and `tc-cdr` of the parent pair. A description of the `tc-` functions appears later.

```
----------------   ----------------
|    |    |    |   |    |    |    |
|    | 0  | 0----->|    | 0  |nil |
|    | |  |    |   |    | |  |    |
-------|--------   -------|--------
       |                  |
       v                  v
----------------   ----------------
|    |    |    |   |    |    |    |
|    | a  | 0  |   |    | b  | 0  |
|    |    | |  |   |    |    | |  |
------------|---   ------------|---
            |                  |
            v                  v
     -----------        -----------
     |    |    |        |    |    |
     |    |aval|        |    |aval|
     |    |    |        |    |    |
     -----------        -----------
```

In this diagram, code slots appear on the left and known values are represented as themselves. When this value is bound to, say, **`store`**, instantiation produces

```
 ----> Store <------ cdr -|
 |     ^              ^   |
 |     |              |   |
 |   --|------------- | --|-------------
 |   | |  |    |    | | | |  |    |    |
 |   | 0  | 0  | 0----->| 0  | 0  |nil |
 |   |    | |  |    | | |    | |  |    |
 |   -------|-------- | -------|--------
car         |        car       |
 ^          v         ^        v
 |   ---------------- | ----------------
 |   |    |    |    | | |    |    |    |
 |-----0  | a  | 0  | |---0  | b  | 0  |
 |   |    |    | |  | | |    |    | |  |
 |   ------------|--- | ------------|---
 |               |    |             |
cdr              v    |             v
 |        ----------- |      -----------
 |        |    |    | |      |    |    |
 |----------0  |aval| |-cdr<---0  |aval|
          |    |    |        |    |    |
          -----------        -----------
```

One advantage of this representation is the sharing of car and cdr operations. The code generator decides, at part of its normal activity, which operations can placed inline, and which must be named by **`let`** expressions.

Primitive Functions

The primitive functions, such as **`+`**, **`cons`**, and **`cdr`** operate on symbolic values. Known values are operated on, while other values cause the creation of a new symbolic value whose code slot indicates the primitive

```
(define pe
  (lambda (exp env stack)
    (cond ((constant? exp) exp)
          ((var? exp) (lookup exp env))
          ((if? exp)
           (let ((pred (pe (if-pred exp) env stack)))
             (cond ((true? pred) (pe (if-then exp) env stack))
                   ((false? pred) (pe (if-else exp) env stack))
                   (else
                     (let ((then-sv (pe (if-then exp) env (mark-stack stack)))
                           (else-sv (pe (if-else exp) env (mark-stack stack))))
                       (generalize then-sv else-sv
                                   (make-if-code pred then-sv else-sv)))))))
          ((lambda? exp) (create-compound-procedure exp env))
          ((call? exp)
           (papply (pe (call-head exp) env stack)
                   (pe (call-arg exp) env stack)
                   stack))
          ((letrec? exp) (pe (letrec-body exp)
                             (recursive-extend-env (letrec-bindings exp) env))))))
```

Figure 4: The specializing interpreter. The variable **stack** is used for termination. This interpreter differs from the standard interpreter in the handling of `if`-expressions and the application of procedures. It also differs in that the closures it constructs contain a *cache* for storing their specializations.

---

function to be called at runtime. The primitive functions also perform type checking, and create code at the appropriate safely level. For example, when `+` is handed **a-value** and 5, it issues the `+` function, but when it is handed **a-natural** and 5, it issues the `tc-+` (TypeChecked-+) function. The `tc-` functions know that their arguments have been typechecked, and do not perform redundant type checking.

The function **cons** always conses together its arguments. When known values are consed together, the code slot contains the pair, otherwise it contains instructions to cons together the arguments at runtime. When handed a pair, both `car` and `cdr` simply return the car or cdr of its value attribute, respectively. They only produce new code when handed **a-value** or **a-list**.

4 The Typing Program Specializer

We now describe the interpreter used by the program specializer (Figure 4). There are two differences between this interpreter and the standard interpreter. First is the presence of the stack, which is used by the termination strategy. The stack and the cache (not shown) are only briefly covered in this paper; they are discussed in detail in [20]. Second, and more importantly, is the fundamental changes to the handling of `if`-expressions and `call`-expressions.

The handler for `if`-expressions first evaluates the predicate. If it evaluates to either *true* or *false*, then the appropriate arm is evaluated. Otherwise, a residual `if`-expression is created, and the corresponding symbolic value represents (at a minimum) all values that could be returned by either arm. The function **generalize** builds this symbolic value by generalizing the symbolic values for each arm. Its code is the residual `if`-expression itself.

In a previous version of the specializer [20] the `if` handler was

```
(define (generalize <v1, c1> <v2, c2> code)
  (cond ((eqv? c1 c2) <v1, c1>)
        ((and (pair? v1) (pair? v2))
         (generalize-pairs v1 v2 code))
        (else <(join v1 v2), code>)))

(define (generalize-pairs (car1 . cdr1) (car2 . cdr2) code)
  (let ((new-car (generalize car1 car2 (make-code 'tc-car code)))
        (new-cdr (generalize p1-cdr p2-cdr (make-code 'tc-cdr code))))
    <(cons new-car new-cdr), code>))

(define (join xval1 xval2)
  (cond ((eqv? xval1 xval2) xval1)
        ((bottom? xval1) xval2)
        ((bottom? xval2) xval1)
        ((top? xval1) xval1)
        ((top? xval2) xval2)
        ((and (xnum? xval1) (xnum? xval2)) 'a-natural)
        ((or (xnum? xval1) (xnum? xval2)) 'a-value)
        etc ...))
```

Figure 5: Code For Performing Generalizations. We have taken the liberty of using psuedo Scheme code that performs pattern matching.

```
(let ((pred (peval (if-pred exp) env stack)))
  (cond ((true? pred) (peval (if-then exp) env stack))
        ((false? pred) (peval (if-else exp) env stack))
        (else
          (let ((then-sv (peval (if-then exp) env (mark-stack stack)))
                (else-sv (peval (if-else exp) env (mark-stack stack))))
==>         (make-sv 'a-value (make-if-code pred then-sv else-sv))))))
```

The marked line indicates the major difference: previously, no type information was computed. To allow the propagation of type information the marked line was changed to

```
(generalize then-sv else-sv (make-if-code pred then-sv else-sv))
```

Generalization maps two symbolic values *s1* and *s2* ito a new symbolic value S representing (at least) all the objects that *s1* and *s2* represent. When *s1* or *s2* are scalars, S's value attribute is the join of the *s1* and *s2*'s value attributes, and the code attribute is determined by the operation requesting the generalization (i.e., it is either a residual `if`-expression or a formal parameter). When both *s1* and *s2* are pairs, their cars are generalized, their cdrs are generalized, and the code slots of the resulting symbolic values take the `tc-car` and `tc-cdr` of S, respectively. Of course, this process is recursive (Figure 5). Only two parts of the specializer call the generalizer, the handler for `if`-expressions, and the termination strategy, which performs generalizations to reduce the number of different values being manipulated by the specializer.

Generalizing the arms of `if` expressions is particularly important when structured values are returned. For example, consider the following (artificial) code:

```
(lambda (x y z)
  (let ((r (if (> x y)
               (list x y z)
               (list x z y))))
    (+ (car r) (caddr r))))
```

```
(define *initial-closure-body* <no-value, no-value>)

(define (specialize-closure cl args)
  (let ((sc (make-specialized-closure (gen-name) arg *initial-closure-body*)))
    (cache! (cons cl args) sc)
    (find-fixed-point cl args sc *initial-closure-body*)))

(define (find-fixed-point cl args sc approx-body)
  (let ((new-body (apply-closure cl args)))
    (set-specialized-closure-body! sc new-body)
    (if (same-type? approx-body new-body)
        sc
        (find-fixed-point cl args sc new-body))))
```

Figure 6: Computing fixed points for the return values of specialized closures. A closure is respecialized until its return type doesn't change.

The specialized program produced for completely unknown inputs is

```
(lambda (x y z)
  (let ((r (if (> x y)
               (list x y z)
               (list x z y))))
    (+ x (tc-caddr r))))
```

Where the elements of `r` can be deduced, they are used (*e.g.*, the value of `r`'s car), and where they cannot be deduced, residual code accessing the value is produced.

5 Type Information from Specialized Closures

Our specializer specializes closures to produce *specialized closures.* (cf Figure 3.) A cache is maintained that keeps all specialized closures indexed by the closure they were specialized from and the arguments they were specialized upon. At each function call, the cache is checked; if there is a hit, a residual call is produced, otherwise the closure is either unfolded or specialized. When specialization occurs, the resulting symbolic value, the closure's arguments, and a freshly created name are collected together into a specialized closure and placed in the cache *indexed* by the arguments. The symbolic value represents not only the code for the specialized closure, but also the values that could be returned by invocations of the specialized closure.

A residual call is represented by a symbolic value. The code slot of the symbolic value is a call to the specialized closure. The type attribute is effectively the type attribute of the specialized closure itself: the symbolic value representing the residual call is constructed by instantiating the symbolic value of the specialized closure with the code representing the function call.

To compute the type (and body) of a specialized closure, the specializer performs a fixed-point computation: the initial approximation to the return result of a closure being specialized is `no-value`, and the closure is reapplied until the returned symbolic value doesn't change (Figure 6). On each iteration, the cache entry is updated with the latest approximation (symbolic value).

The simplicity of our method may seem surprising to those who have implemented fixed-point analysis over structured values for Binding Time Analysis. Where are our guarantees of finite domains? Where are our descriptions of data structures? What are our abstractions? We respond that we have an advantage in being an online specializer: rather than describe the possible data that may occur during specialization, we simply construct the data. Termination of specialization guarantees finiteness of data structures, as only infinite processes can construct infinite data structures.

Theorem 1 *(Termination of Fixed Point Iteration) Assuming that specialization terminates, then the fixed point iteration does so as well.*

Outline of Proof: At each iteration, the input to the closure being specialized does not change; only the specialized closure changes. The specialized closure's value attribute, which represents the return type, is no larger than its code attribute. Therefore, assuming that the specialized code does not grow on each iteration, the monotonicity of the iteration guarantees termination of the iterations. We claim that any computation that would cause the specialized closure to grow unboundedly on iteration to iteration would have caused the specializer to not terminate anyway. Remember that on each iteration the input does not change and the specializer goes through the same steps each time; the only thing that changes is the approximation to the return type of residual calls. Therefore, for the code attribute to continually grow, there must be some computation that generates an unbounded amount of output regardless of the input. But such a computation would be run during specialization, causing non-termination of the specializer. •

Note that termination of the specializer on any given iteration does not guarantee termination on subsequent iterations. On each iteration, a given function may see a different symbolic value, and one of the symbolic values may cause the function to loop. For example, suppose that on one of the iterations, is shown that a specialized closure returns a list containing 5 and an unkown value. The known value 5 may cause non-termination on the next iteration. We have discovered that, in practice, the fixed-point is computed within 2 or 3 iterations.

As an example, consider again the MP interpreter fragment (Figure 2). When compiling a `while` statement the specializer must make specialized version of the loop

```
(let while-loop ((store store))
  (if (mp-exp text-exp store)
      (while-loop (mp-command while-com store))
      store)).
```

Assume that `store` has the shape (type) `((a . ?) (b . ?))` and the specializer has decided to specialize `while-loop`. The first action taken by the specializer is to add an entry to `while-loop`'s cache with an initial type approximation of `no-value` and name `while-loop32`. It then applies its body (the `if` expression) to the store. Evaluating the antecedent yields an unknown symbolic value, so both arms are specialized. Because of a cache hit, the specialization of the consequent yields a residual call to `while-loop32` with type `no-value`. (The cache is hit assuming that the type of the specialization of `mp-command` equals the type of `store`.) The type of the residual `if` expression is the generalization of `no-value` and `((a . ?) (b . ?))`, which is `((a . ?) (b . ?))`, becomes the return type of `while-loop32`. Because there was a change in type, `while-loop` is applied again to `store`. The only change is in the return type of the residual call, which is now `((a . ?) (b . ?))`. This type becomes the type of the residual `if`, and of `while-loop32`, terminating the process.

The complexity of a naive implementation may be unnecessarily large for nested recursive functions, because fixed-points would be performed exponentially. The problem is that within each loop, a new closure representing a subloop would be created, and we would lose the previously found fixed-point because the cache is indexed by closures. If the complexity of finding fixed-points is too large, one could use to methods that would allow greater reuse of previous specializations. For example, [16] describes a method of indexing specializations that does just this, and speeds up fixed-point analysis by an order of magnitude.

6 Abstract Interpretation with Concrete Values

When the specialized program is first order, there is a strong relationship between specialization as presented in this paper and abstract interpretation. The two major differences are: 1) abstract interpretation is guaranteed to terminate (liveness) whereas specialization isn't guaranteed to terminate, and 2) in abstract interpretation all concrete values are abstracted into the abstract domain before abstract interpretation begins, whereas specialization only abstracts as necessary to achieve termination. Therefore, if one always gave well behaved programs to the specializer, or bounded the running of specializer to ensure termination (as [21] does in its type evaluator), then, by embellishing the type system as needed, what we term *abstract interpretation with concrete value* could be performed: abstract interpretation that allows for concrete values during abstract interpretation.

Consider the standard example of determining whether (a fragment of) a program returns a positive number, 0, or a negative number. Abstract interpretation with concrete values can decide this, as well as return a concrete value when possible. For example, consider the program P,

```
(lambda (x y z)
  (if (> x y)
      (list x (+ y -5) (+ x 20) z)
      (list y 9 25 0))).
```

Assuming that the proper changes have been made to the primitive functions (e.g., `+`, `-`) and to the type system, then the return "type" of `(TSP P 5 (a-positive-natural) (a-negative-natural)` would be `(a-positive-natural a-natural 25 a-nonpositive-natural)`. Because data dependencies are considered during abstract interpretation with concrete values, changing the meanings of $>$ and $<$ to handle positive and negative numbers would be worthwhile.

Similarly, strictness analysis is performed by simply supplying `a-values` (possibly terminates) and `no-value` (doesn't terminate) as arguments. If the result symbolic value is `no-value`, then the function is strict in the argument that was `no-value`. (When using specialization for strictness analysis, code generation could be disabled.)

Obviously, these ideas need fleshing out. One future experiment we wish to conduct is performing an extremely accurate binding time analysis using these ideas. Such an approach would automatically perform a multivariant BTA [15], which is very important for maintaining accuracy in an offline automatic program specializer.

7 Conclusions and Future Research

We have presented an online program specializer that uses symbolic values to both represent a value and the code for creating the value. By introducing generalization, we were able to derive and use type information from residual if-expressions. By iterating the creation of specialized closures, we were able to derive and use type information about specialized recursive closures.

One benefit is more accurate specialization. A larger class of programs can now be written in a natural style without the partial information losing information. A second benefit is abstract interpretation with abstract values, which we have just begun to investigate.

On problem that needs solving is dead code elimination on data structures whose slots are not accessed by the specialized program. Our methods cause some slots to become useless, but they are still constructed at runtime. We want to investigate a combination of CPS conversion and *arity raising* [18]. These techniques are important for further boosting the efficiency of specialized programs. For example, the specialized program presented in Appendix C would be helped by CPS conversion plus arity raising. This would eliminate those parts of the store that are no longer needed, and disperse what's left of the store into separate variables.

A MP Interpreter

The MP language is a simple imperative language with four commands (`:=`, `if`, `while`, and `begin`). Each program declares its inputs and local variables. At the start of interpretion, the input parameters are bound to the inputs, and the local variables are bound to `nil`. The result of an MP program is the final store.

This code was derived from the code provided by Mogensen [14] who, in turn adapted it from Sestoft. This interpreter differs from Mogensen's in that his interpreter was written in CPS style (possibly to avoid the specific problem that this research solves), whereas this code is written in a direct recursive descent style. Another difference is that his interpreter has programmer annotations for directing its partial evaluation (automatic termination methods do not yet handle the coding style his interpreter was written in).

```
(define mp-interp
  '(lambda (program input)
    (letrec
     ((INIT-STORE
       (lambda (parms vars input)
         (if (null? parms)
             (map (lambda (x) (cons x '())) vars)
             (let ((new-binding (cons (car parms) (car input))))
               (cons new-binding (init-store (cdr parms) vars (cdr input)))))))
      (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))
            ((eq? token 'begin)
             (let begin-loop ((coms rest) (store store))
               (if (null? coms)
                   store
                   (begin-loop (cdr coms) (mp-command (car coms) store)))))
            ;; when the program is overgeneralized, this is hit.
            ;; we return the store rather than signal an error to make types work.
            ('#t store)))))
      (MP-WHILE
       (lambda (com store)
         (if (mp-exp (cadr com) store)
             (mp-while com (mp-command (caddr com) store))
             store)))
      (MP-EXP
       (lambda (exp store)
         (if (symbol? exp)
             (lookup exp store)
             (let ((token (car exp)) (rest (cdr exp)))
               (cond ((eq? token 'quote) (car rest))
                     ((eq? token 'car) (car (mp-exp (car rest) store)))
                     ((eq? token 'cdr) (cdr (mp-exp (car rest) store)))
                     ((eq? token 'atom) (not (pair? (mp-exp (car rest) store))))
                     ((eq? token 'cons) (cons (mp-exp (car rest) store)
                                              (mp-exp (cadr rest) store)))
                     ((eq? token 'equal) (eq? (mp-exp (car rest) store)
                                              (mp-exp (cadr rest) store)))
                     ('#t 'error2))))))
```

```
 (UPDATE
  (lambda (store var val)
    (let update-loop ((store store))
          (IF (eq? (caar store) var)
              (cons (cons (caar store) val) (cdr store))
              (cons (car store) (update-loop (cdr store)))))))
 (LOOKUP
  (lambda (var store)
    (if (eq? (caar store) var)
        (cdar store)
        (lookup var (cdr store))))))
(let ((parms (cdr (cadr program)))
      (vars (cdr (caddr program)))
      (main-block (cadddr program)))
  (mp-command main-block (init-store parms vars input))))))
```

B (pp (graph->scheme (p mp-interp (a-value) (a-value))))

This is the MP Interpreter specialized without constraint, which inlines both non-recursive and recursive functions. For example, the **lookup** and **BEGIN** loops have been moved to where they are called. The **delay** and **force** expressions were introduced by the code generator when it hoisted the conditionalized invariant expression **(mp-exp-36.1 (cadr T59) store-18)** out of a loop. The **delay** is needed because although the expression is invariant, the code generator cannot prove that its value will be requested during the loop.

```
(lambda (program-2 input-1)
 (letrec
  ((mp-command-20.1
    (lambda (com-19 store-18)
     (let
      ((T59 (cdr com-19)))
      (letrec
       ((update-loop-62.1
         (let*
          ((T233 (car T59)) (T223 (delay (mp-exp-36.1 (cadr T59) store-18))))
          (lambda (store-61)
           (if (eq? (caar store-61) T233)
               (cons (cons (caar store-61) (force T223)) (cdr store-61))
               (cons (car store-61) (update-loop-62.1 (cdr store-61)))))))
        (mp-while-44.1
         (lambda (com-43 store-42)
          (if (mp-exp-36.1 (cadr com-43) store-42)
              (mp-while-44.1 com-43 (mp-command-20.1 (caddr com-43) store-42))
             store-42)))
        (mp-exp-36.1
         (lambda (exp-35 store-34)
          (if (symbol? exp-35)
              (letrec
               ((lookup-40.1
                 (lambda (var-39 store-38)
                  (if (eq? (caar store-38) var-39)
                      (cdar store-38)
                      (lookup-40.1 var-39 (cdr store-38))))))
               (lookup-40.1 exp-35 store-34))
             (let* ((T106 (car exp-35)) (T93 (cdr exp-35)))
              (cond
               ((eq? T106 'quote) (car T93))
               ((eq? T106 'car) (car (mp-exp-36.1 (car T93) store-34)))
               ((eq? T106 'cdr) (cdr (mp-exp-36.1 (car T93) store-34)))
               ((eq? T106 'atom) (tc-not (pair? (mp-exp-36.1 (car T93) store-34))))
               ((eq? T106 'cons) (cons (mp-exp-36.1 (car T93) store-34)
                                       (mp-exp-36.1 (cadr T93) store-34)))
               ((eq? T106 'equal) (eq? (mp-exp-36.1 (car T93) store-34)
                                       (mp-exp-36.1 (cadr T93) store-34)))
               (else 'error2)))))))
       (let ((T73 (car com-19)))
        (cond
         ((eq? T73 ':=) (update-loop-62.1 store-18))
         ((eq? T73 'if) (if (tc-not (null? (mp-exp-36.1 (car T59) store-18)))
                           (mp-command-20.1 (cadr T59) store-18)
                           (mp-command-20.1 (caddr T59) store-18)))
         ((eq? T73 'while) (mp-while-44.1 com-19 store-18))
         ((eq? T73 'begin) (letrec
                            ((begin-loop-59.1
                               (lambda (coms-58 store-57)
                                (if (null? coms-58)
                                   store-57
                                   (begin-loop-59.1
                                    (cdr coms-58)
                                    (mp-command-20.1 (car coms-58) store-57))))))
                          (begin-loop-59.1 T59 store-18)))
       (else store-18))))))))
(mp-command-20.1
 (cadddr program-2)
 (letrec
  ((init-store-8.1
    (lambda (parms-7 vars-6 input-5)
     (if (null? parms-7)
         (letrec
          ((map-14.2
            (lambda (l-13)
             (if (null? l-13)
                 '()
                 (cons (list (car l-13)) (map-14.2 (cdr l-13)))))))
          (map-14.2 vars-6))
         (cons (cons (car parms-7) (car input-5))
               (init-store-8.1 (cdr parms-7) vars-6 (cdr input-5)))))))
  (init-store-8.1 (cdadr program-2) (cdaddr program-2) input-1)))))
```

C (pp (graph->scheme (p mp-interp mp-compare (a-value))))

The following is a comparison program written in the MP language.

```
(define mp-compare
  '(program (pars a b) (dec flag out)
    (begin
      (:= flag '#t)
      (while flag
        (if a
            (if b
                (begin (:= a (cdr a))
                       (:= b (cdr b)))
                (begin (:= out 'a)
                       (:= flag '())))
            (if b
                (begin (:= out 'b) (:= flag '()))
                (begin (:= out 'ab) (:= flag '()))))))))
```

On the next page is the specialization of the MP Interpreter on the **MP-COMPARE** program. Note the presence of the TC functions in the main loop. These, and the lack of residual variable lookup code, are possible because of the typing done by the specializer. Without this typing, this code would not be nearly as efficient as it is. Note that the specialized program does not simply construct the store and immediately pass it to the loop. Instead, it first does some of the work of the loop, and then passes the store to the loop. The reason for this is that the original store binds both **out** and **flag** to **nil**, so this extra information is exploited outside of the loop. The loop is specialized for a store where none of the bindings are known.

```
(lambda (input-1)
 (letrec
  ((mp-while-10.1
    (lambda (store-8)
     (let* ((T60 (tc-cdr store-8)) (T87 (tc-cdr T60)))
      (if (tc-cdar T87)
          (mp-while-10.1
           (let* ((T66 (tc-car store-8))
                  (T95 (tc-cdr T66))
                  (T62 (tc-car T60))
                  (T75 (tc-cdr T62)))
            (cond
             ((tc-not (null? T95))
              (if (tc-not (null? T75))
                  (list* (cons 'a (cdr T95)) (cons 'b (cdr T75)) T87)
                  (list* T66 T62 '((flag) (out . a)))))
             ((tc-not (null? T75)) (list* T66 T62 '((flag) (out . b))))
             (else (list* T66 T62 '((flag) (out . ab)))))))
          store-8)))))
  (mp-while-10.1
   (let* ((T13 (car input-1))
          (T6 (cadr input-1))
          (T16 (cons 'a T13))
          (T9 (cons 'b T6)))
    (cond
     ((tc-not (null? T13))
      (if (tc-not (null? T6))
          (list* (cons 'a (cdr T13)) (cons 'b (cdr T6)) '((flag . #T) (out)))
          (list* T16 T9 '((flag) (out . a)))))
     ((tc-not (null? T6)) (list* T16 T9 '((flag) (out . b))))
     (else (list* T16 T9 '((flag) (out . ab)))))))))
```

References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Halstead Press, 1987.

[2] L. Beckman et al. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7(4):291–357, 1976.

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

[5] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer Magazine*, 1990. (to appear).

[6] 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.

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

[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. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[10] N. D. Jones. Binding time analysis and the taming of self application. *ACM Transactions on Programming Languages and Systems*, 1990. (to appear).

[11] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[12] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, 1986.

[13] J. Launchbury. Projections for specialization. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 299–315. North-Holland, 1988.

[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 Copenhagen, Copenhagen, Denmark, March 1989.

[16] E. Ruf and D. Weise. Using types to avoid redundant specialization. submitted to 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, November 1990.

[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] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

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

[20] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online program specialization. submitted to 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, November 1990.

[21] 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.