COMPUTING TYPES DURING PROGRAM SPECIALIZATION

Daniel Weise
Erik Ruf

Technical Report: CSL-TR-90-441

October 1990

This research has been supported in part by NSF Contract No. MIP-8902764, and in part by Advanced Research Projects Agency, Department of Defense, Contract No. N00014-87-K-0828. Erik Ruf is funded by an AT&T Foundation Ph.D. Scholarship.

Computing Types During Partial Evaluation

Daniel Weise and Erik Ruf

Technical Report: CSL-TR-90-441
(Also FUSE Memo 90-3)

October 1990

Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, California 94305

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. The specializer exploits type information to increase the efficiency of specialized functions. This research has two benefits, one anticipated and one unanticipated. The anticipated benefit is that programs that are to be specialized can now be written in a more natural style without losing accuracy during specialization. The unanticipated benefit is the creation of what we term ***concrete abstract interpretation.*** This is a method of performing abstract interpretation with concrete values where possible. The specializer abstracts values as needed, instead of requiring that all values be abstracted prior to abstract interpretation.

Key Words and Phrases: Program Specialization, Partial Evaluation, Types, Abstract Interpretation.

Introduction

A ***program specializer*** (also called a ***partial evaluator)*** transforms a program and a description of the possible inputs of the program into a program specialized for those inputs. For example, a two argument function could be specialized for its first argument being 2 and its second argument being any value at all. Program specialization is an experimentally proven technique for creating compilers and compiler generators [13], automatically rederiving important algorithms [8], speeding up computations by two orders of magnitude [5], parallelizing scientific code [4], and optimizing programs [6].

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.[1] 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.) The previous specializers that have computed type information [10, 20] did not compute the types of specialized subprograms, and [10] 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 [15]) 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 (nil versus a pair) and dispatching. Our system does compile away this extraneous type checking and dispatching that even typed languages such as ML must defer until runtime. (Correctly staged programs can get this benefit, as well as others, through arity raising. This research offers this benefit for all programs, not just correctly staged ones.)

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

[1] **A note on terminology. We say that a program specializer *specializes* programs, or, alternatively, produces *specialized programs.* We tend to apply the adjective *residual* to expressions, not to programs. Residual expressions are the constituents of specialized programs.**

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.

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 have been presented before to remedy other problems in program specialization such as code duplication, type propagation, and overly strict interpretation of values versus code [24] ([5] uses *placeholders*, which are very similar to symbolic values). The other two mechanisms are a novel contribution to program specialization. We will see that the important new mechanism is generalization, and with that in place, performing fixed-point iteration is straightforward.

This research allows for ***concrete abstract interpretation*** (CAI). 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 CAI is directly related to that of the program specializer itself.

Binding Time Analysis (BTA), an important tool for constructing self-applicable specializers [13, 19, 9, 12], 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 little 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*** [18, 9], 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.

This research is closely related to that of Young and O'Keefe [26], 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

```
(define (update-store store name new-value)
  (if (eq? (caar store) name>-
      (cons (cons (caar store) new-value) (cdr store))
      (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.

---

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 is synergistically used in the termination strategies.

Haraldsson's REDFUN-2 [10] was a redesign of [3] that was designed to compute and track type information during partial evaluation. 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 fully automatic, as it relies on programmer advice to carry out its operations.

Also of extreme relevance is the work of Schooler [20], 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, 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 [24]. 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 7 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,*** which are the lynchpin of our method (the result of partially evaluating an expression is always a symbolic value). The specializer and generalization are presented in the fourth section. The fifth section outlines the specializer's termination strategies. Fixed-point iterations are presented in Section 6. Finally, Section 7 discusses concrete abstract interpretation.

1 Program Specialization

The user of a specializer specifies the classes of values the program is to be specialized for. The class 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 υ. Different specializers allow different classes to be described, but they all share the same approach: the domain[2] 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 the representation of a program in $\mathcal{L}$, V be value domain of $\mathcal{L}$, T[] be a function mapping representations of consistent subsets of V into such subsets,*** sarg ***be the representation of a subset of V, and let E[] map programs to their meanings. A*** strict program specializer ***is a program SP mapping $\mathcal{L}$*** x ***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. 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.

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

Our definition of program specialization subsumes that of Jones [11], who describes a partial evaluator as a program that accepts ***part of*** a program's inputs, and that of Launchbury [17] 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. There is little fundamental difference between Launchbury's approach and ours. We see the distinction between our formulation and his the same as the distinction between the standard formulation of Abstract Interpretation and the projections formulation of Abstract Interpretation [23]. We simply find our approach more natural. When we consider Mixed Computation, the differences will become more pronounced.

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 the 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 the range of the specialized program.

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

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

zA ***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 [2] and to the definition of [26]:

Definition 4 ***(Safety of Typing Program Specialization) Assuming the conditions of Definition 1, 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 known value is within the type computed by the specializer. We state without proof that our specializer is safe.

2 Syntax and Standard Semantics

This paper considers a simple untyped eager functional programming language operating over numbers, booleans, pairs, symbols, and nil. For simplicity of exposition we restrict functions to accept only one argument. The abstract syntax of the language is:

i	∈	***Id***	Identifiers
e, body	∈	***Exp***	Expressions
n	∈	***Num***	Numerals
b	∈	`t I f`	Boolean constants

```
e    ::= i | n | b |
         (if el e2 e3) |           Two armed conditional
         (lambda i body) |         Lambda Abstraction
         (call el e2) |            Function Application
         (letrec (i e)* body)      Mutually Recursive Definitions
```

```
(define (eval exp env)
  (cond ((boolean? exp) exp)
        ((nat? exp) exp)
        ((var? exp) (lookup exp env))
        ((if? exp) (if (eval (if-pred exp) env)
                       (eval (if-then exp) env)
                       (eval (if-else exp) env)))
        ((lambda? exp) (create-closure exp env))
        ((call? exp) (apply (eval (call-head exp) env)
                            (map (lambda (exp) (eval exp env))
                                 (call-args exp))))
        ((letrec? exp) (eval (letrec-body exp)
                             (recursive-extend-env (letrec-bindings exp) env)))))

(define (apply f args)
  (cond ((primitive-proc? f) (apply-primitive-proc f args))
        ((closure? f) (apply-closure f args))))

(define (apply-closure (cl-formals cl-body cl-env) args env )
  (eval cl-body (extend-env cl-env cl-formals args)))
```

Figure 2: Evaluator fragment for the source Language. Our notation allows destructuring (pattern matching) during binding.

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 standard semantic domains and function types are

$$\begin{array}{rcl}
\mathbf{\textit{Nat}} & = & \text{the flat domain of natural numbers} \\
\textit{Bool} & = & \mathbf{\textit{true}} + \mathbf{\textit{false}} \\
\mathbf{\textit{Pair}} & = & \mathbf{\textit{VAL}} \times \mathbf{\textit{VAL}} \\
\textit{Nil} & = & \mathbf{\textit{nil}} \\
\textit{Func} & = & \textit{VAL} \mapsto \mathbf{\textit{VAL}} \\
\mathbf{\textit{VAL}} & = & \mathbf{\textit{Nat}} + \textit{Bool} + \mathbf{\textit{Pair}} + \textit{Nil} + \textit{Func}
\end{array}$$

Rather than provide semantic equations, we provide the skeleton of an interpreter based upon the interpreters discussed in [1]. When we later present the program specializer, we will present a variation on this interpreter.

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

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

Figure 3: Domains used by the program specializer when interpreting programs. Note that the partial evaluator 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 *SVALs* rather than ***Code,*** but it is easier to describe the system using just ***Code*** elements. The implementation keeps *SVALs* 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.

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 (v, c).

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 [24], so a code generator is used to express the code in some particular programming language.

Example symbolic values include:

$\langle 4,4\rangle$	The number 4
(a – natural, a)	A number whose name is a
$\langle(\langle 4,4\rangle.\langle$a – natural, a)), (cons4a)$\rangle$	The cons of the above ***2*** items.
$\langle(\langle$a – boolean, (car(f a))).(a – natural, (cdr(f a)))), (f a))	A pair that will be created by a runtime call to f.

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

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

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

There is a parallel between the a-natural, a-boolean, no-value, and a-value, and the do-***main*** objects $\top_{nat}$, $\top_{bool}$, $\perp_{VAL}$, 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. 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. [3] The symbolic values so constructed have empty code slots. The specializer ***instantiates*** the symbolic values before starting the symbolic execution. Instantiation initializes the code slots of the provided symbolic values. Instantiation makes the code slots of unknown scalar values be the formal parameter the symbolic value is bound to. For pairs, it operates recursively, making the code slots of the car and cdr of a symbolic pair take the car and cdr of the symbolic pair itself. [4] Instantiation is also invoked when computing the return value of residual function calls (Section 5).

[3]There is no provided function for building the equivalent of no-value. This value is created 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 7).

[4]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.

```
(define pe-+
  (lambda (x y)
    (let ((x-val (sv-val x)) (y-val (sv-val y)))
      (cond ((no-value? x) x)
            ((no-value? y) y)
            ((and (xnum? x-val) (xnum? y-val))
             (if (and (known-value? x) (known-value? y))
                 (known-value (+ x-val y-val))
                 (make-sv 'a-natural (make-code 'tc-+ x y))))
            ((and (possible-xnum? x) (possible-xnum? y))
             (make-sv (join x-val y-val) (make-code '+ x y)))
            (else (type-error "Wrong type arguments to +"))))))

(define pe-cons
  (lambda (x y)
    (if (and (known-value? x) (known-value? y))
        (make-sv (cons x y) (cons x y))
        (make-sv (cons x y) (make-code 'cons x y)))))

(define pe-car
  (lambda (x)
    (cond ((no-value? x) x)
          ((pair? (sv-val x)) (car sv-val x))
          ((possible-pair? (sv-val x)) (make-sv 'a-value (make-code 'car x)))
          (else (type-error "CAR needs a pair.")))))
```

Figure 4: The code for +, car, and cdr. Predicates such as possible-xnat? return true if their argument is an element of the domain, or higher up in the lattice. When handed a pair, car and cdr always operate on the value attribute of their argument. They only produce new code when handed a-value.

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 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 (Figure 4). 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.

```
(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-code 'if (list 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 5: The specializing interpreter. The variable stack is used for termination. The global variable cache is not shown here. It is used to remember specialized closures. This interpreter differs from the standard interpreter in the handling of if-expressions and the application of procedures.

4 The Typing Program Specializer

We now describe the interpreter used by the program specializer (Figure 5). 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 [25]. Second, and more importantly, is the fundamental changes to the handling of if-expressions and call-expressions (Section 5).

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. Generalization is particularly useful 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 (generalize <v1, cl> <v2, c2> code)
  (cond ((eqv? cl 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 pl-cdr p2-cdr (make-code 'tc-cdr code))))
    <(cons new-car new-cdr), code>))

(define (join xvall xva12)
  (cond ((eqv? xvall xval2) xvall)
        ((bottom? xvall) xva12)
        ((bottom? xval2) xvall)
        ((top? xvall) xvall)
        ((top? xva12) xval2)
        ((and (xnnm? xvall) (xnnm? xva12)) 'a-natural)
        ((or (xnum? xvall) (xnnm? xva12)) 'a-value)
        etc . ..))
```

Figure 6: Code For Performing Generalizations. We have taken the liberty of using psuedo Scheme code that performs pattern matching.

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 (the value of r's car), and where they cannot be deduced, residual code accessing the value is produced.

Generalization[5] 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 6). 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. Despite its simplicity, or maybe because

[5]**A note on terminology. Turchin [22] introduced the term *generalization* for the process of losing information about a value. Other researchers researchers also use this term. We prefer to use the term abstraction when a single argument is mapped higher up in the lattice, (or to a simpler lattice). We reserve the term generalization for when two arguments are mapped upwards in the lattice. Abstraction is a unary operation, while generalization is a binary operation.**

```
(define papply
  (lambda (f args stack)
    (cond ((primitive-procedure? f) (primitive-apply f args))
          ((lookup-specialization-cache f args)
           (make-residual-call (lookup-specialization-cache f args) args))
          ((specialize? f args stack)
           (let ((stack-and-arg (generalize-argument f args stack)))
             (let ((sf (specialize-closure f (car stack-and-erg) (cdr stack-and-arg))))
               (make-residual-call sf args))))
          (else (apply-closure f args stack)))))

;; The real version of this appears in a later figure.
(define specialize-closure
  (lambda (cl arg stack)
    (let ((sc (make-specialized-closure
                (gen-name)
                arg
                (apply-closure cl arg stack))))
      (cache! (cons cl arg) sc)
      sc)))

(define apply-closure
  (lambda (cl args stack)
    (let (((cl-formals cl-body cl-env) cl))
      (peval cl-body
             (extend-env cl-formals arg cl-env)
             (cons (cons cl args) stack)))))

(define make-residual-call
  (lambda ((sc-name sc-args sc-result) args)
    (instantiate sc-result
                 (make-code sc-name args))))
```

Figure 7: Application of compound procedures. Termination and specialization decisions are made here. Though not discussed in this paper, our partial evaluator has excellent termination properties [25], and terminates more often than other published methods.

of it, generalization is very powerful. Performing fixed-point iterations is straightforward once generalization is in place.

5 Papply: Termination and Specialization

Termination in nearly all automatic program specializers is based upon some form of loop detection. Loops are detected when applying closures. 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 (Figure 7). Program specializers have different strategies for deciding when to specialize a closure. Some specialize for all functions calls (e.g., [14]), and some specialize only when recursive loops are detected (e.g., [25]). There is little difference between unfolding a closure and specializing it: in each case the closure is applied to its arguments. 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. 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.

For program specialization to terminate, every possible combination of functions and arguments must make it into the cache[6] and there must only a finite number of cache entries (otherwise an infinite number of specializations will be generated) [11]. When there are an infinite number of possibilities, this will not occur. Therefore program specializers resort to abstraction to reduce the number of values generated during specialization. It is common to abstract values to just the one-point domain unknown (which corresponds to our use of a-value). Loop detection and specialization is then based upon the abstracted values, rather than the concrete values. The accuracy and termination properties of a program specializer directly depend upon when and how abstraction is done. Too much abstraction, and accuracy suffers, too little abstraction, and termination suffers [25].

Our specializer remembers all active procedures and the arguments they were called on, as well as the predicates of if-expressions that did not evaluate to ***true*** or *false*, to decide whether to unfold or specialize a call. When it decides to specialize, it does so upon arguments that are the generalization of the current call and some previous call such that the least amount of generalization is performed (the least amount of information is lost). The code attribute of the generalized value is the formal parameter of the function being specialized. The initial approximation to the return value of the specialized closure is no-value (I).

6 Type Information from Specialized Closures

The major machinery for approximating the return results of recursive functions is already in place, namely, generalization of symbolic values. All that remains is to perform a fixed point iteration when creating a specialized closure. The fixed point is found in the usual way: 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 8). On each iteration, the cache entry is updated with the latest approximation (symbolic value).

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 at most a constant factor 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 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. •

[6]More precisely, only a finite number of occurences of them can escape being placed in the cache.

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

Note that termination of the specializer on any given iteration does not guarantee termination on subsequent iterations. On each iteration, a given function will 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. In practice, the fixed-point is computed within 2 or 3 iterations.

The complexity of a naive implementation may be unnecessarily large for nested recursive functions, because fixed-points would be exponentially performed. 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 resort to engineering fixes that would allow cache hits to occur and thereby avoid recomputing fixed-points.

7 Concrete Abstract Interpretation

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 value are abstracted into the abstract domain before abstract interpretation begins, whereas specialization only abstracts as necessary to achieve termination. Therefore, if one were to bound the running of specializer to ensure termination (as [26] does in its type evaluator), or always gave well behaved programs to the specializer, then, by embellishing the type system as needed, what we term ***concrete abstract interpretation*** could be performed: abstract interpretation that allowed 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. Concrete abstract interpretation 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 concrete abstract interpretation, 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. For example, we have not indicated how a sticky interpretation would fit in this framework. 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 [19], which is very important for maintaining accuracy in an offline automatic program specializer.

8 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 concrete abstract interpretation, 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*** [21]. 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.

Those who want to experiment with the specializer (which we call FUSE) may FTP the file ftp/dist/popl-fuse.tar from **sid.stanford.edu.** The implementation is, of course, more complex than that described in this paper, but reproducing any of the examples in this paper should be simple. If you take a copy of FUSE, please send email informing us (ruf Qdolores. stanf ord. edu, daniel@mojave.stanford.edu) that you have done so.

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 [18] 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 (corn store)
         (let ((token (car corn)) (rest (cdr corn)))
           (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 corn store))
            ((eq? token 'begin)
             (let begin-loop ((corns rest) (store store))
               (if (null? corns)
                   store
                   (begin-loop (cdr corns) (mp-command (car corns) 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 (corn store)
         (if (mp-exp (cadr corn) store)
             (mp-while corn (mp-command (caddr corn) 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-l)
 (letrec
  ((mp-command-20.1
    (lambda (corn-19 store-18)
     (let
      ((T59 (cdr corn-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 (corn-43 store-42)
          (if (mp-exp-36.1 (cadr corn-43) store-42)
              (mp-while-44.1 corn-43 (mp-command-20.1 (caddr corn-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 corn-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 corn-19 store-18))
      ((eq? T73 'begin) (letrec
                         ((begin-loop-59.1
                            (lambda (corns-58 store-57)
                             (if (null? corns-58)
                                 store-57
                                 (begin-loop-59.1
                                  (cdr corns-58)
                                  (mp-command-20.1 (car corns-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)
                 '0
                 (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-l)))))
```

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 JO>>>
            (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-l)
 (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-lo.1
   (let* ((T13 (car input-l))
          (T6 (cadr input-l))
          (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] Abelson, Hal, Sussman, Gerald, with Sussman, Julie, ***The structure and interpretation of Computer Programs,*** MIT Press, 1984.

[2] Abramsky, Samsom, and Hankin, Chris, ***Abstract Interpretation of Declarative Languages,*** Halstead Press, 1987.

[3] Beckman, L., ***et. al.,*** "A partial evaluator, and its use as a programming tool," ***Artificial*** *Intelligence* Vol 7, Number 4, pps. 319-357, 1976.

[4] Berlin, Andrew, ***A Compilation Strategy for Numerical Programs Bused on*** *Partial* ***Evaluation,*** Masters Thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory Technical Report TR-1144, Cambridge, MA, July 1989.

[5] Berlin, Andrew, and Weise, Daniel, "Compiling Scientific Programs using Partial Evaluation," Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo AIM-1145, Cambridge, MA, July 1989.

[6] Berlin, Andrew, "Partial Evaluation Applied to Numerical Computation," to appear the ***Proceedings*** *of* ***the 1990 ACM Conference on Lisp and Functional Programming,*** Nice, France, 1990.

[7] Bjørner, D., Ershov, A. P., and Jones, N. D., (eds.), ***Partial Evaduution and*** *Mixed* ***Computation,*** North Holland, 1988.

[8] Consel, Charles, and Danvy, Olivier, "Partial evaluation of pattern matching in string," ***Information Processing Letters,*** vol 30, No 2, pps. 79-86, 1989.

[9] Consel, Charles, "Binding time analysis for higher order untyped functional languages," ***Proceedings*** *of* ***the 1990 ACM Conference on Lisp and Functional Programming,*** Nice, France, pps. 264-272, 1990

[10] Haraldsson, Anders, ***A Program*** *Manipulation* ***System Bused on Partial Evaluation,*** Linkoping Studies in Science and Technology Dissertation No. 14, Linkoping University, 1977.

[11] Jones, N. D., "Automatic program specialization: A re-examination from basic principles," pps. 225-282 of [7].

[12] Jones, N. D., ***et. al.,*** "Binding time analysis and the taming of self application," to appear in TOPLAS, ACM.

[13] Jones, N. D., Sestoft, P., and Sondergaard, "Mix: A self-applicable partial evaluator for experiments in compiler generation ," Vol 1, Nos 3/4, International Journal of Lisp and Symbolic Computation, Kluwer Publishers, 1988.

[14] Jones, N. D., Sestoft, P., and Sondergaard, "An experiment in partial evaluation: The generation of a compiler generator ," In J. P. Jouannaud (ed.): ***Rewriting Techniques and Applications,*** *Dijon,* ***France,*** 1985, LNCS 202, pps 124-140, 1985.

[15] Jorring, U. and Scherlis, W. L., "Compilers and staging transformations," In ***Thirteenth A CM Symposium on Principles*** *of* ***Programming Languages,*** St. Petersburg, Floida, pps 86-96, 1986.

[16] Kahn, Kenneth M., "A partial evaluator of Lisp programs written in Prolog," In M. Van Caneghem (ed.): ***First International Logic Programming Conference, Marseille, France 1982,*** pps 19-25, 1982.

[17] Launchbury, John, "Projections for Specialization," in [7].

[18] Mogensen, Torben, "Partially Static Structures," pps. 325-347 of [7].

[19] Mogensen, Torben, ***Binding Time Aspects** of **Partial Evaluation,*** PhD Thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, March 1989.

[20] Schooler, R., ***Partial Evaluation as a means** of **Lunguage Extensibility,*** Masters Thesis, 84 pages, MIT/LCS/TR-324, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1984.

[21] Sestoft, Peter, "Automatic call unfolding in a partial evaluator," in [7], pps. 485-506, 1988.

[22] Turchin, Valentin, "The concept of a supercompiler," ***Transactions on Programming Languages and Systems,*** 8(3), pps. 292-325, 1986.

[23] Wadler, P., and Hughes, R. J. M., "Projections for Strictness analysis," Proceedings of the ***Third International Conference on Functional Programming Languages and Computer Architecture,*** Portland, Oregon, September 1987.

[24] Weise, Daniel, "Graphs as an intermediate representation for partial evaluation," CSL-TR-90-421, Computer Systems Laboratory, Stanford University, CA, 1990.

[25] Weise, Daniel, "Termination versus Accuracy in Automatic Partial Evaluators," in preparation.

[26] Young, Jonathan, and O'Keefe, Patrick, "Experience with a type evaluator," pps. 573-581 of [7].