Automatic Online Partial Evaluation*

Daniel Weise
Roland Conybeare
Erik Ruf
Scott Seligman

Computer Systems Laboratory
Margaret Jacks Hall
Stanford University
Stanford, CA 94305-2140

July 1, 1991

Abstract: We have solved the problem of constructing a fully automatic online program specializer for an untyped functional language (specifically, the functional subset of Scheme). We designed our specializer, called *Fuse*, as an interpreter that returns a trace of suspended computations. The trace is represented as a graph, rather than as program text, and each suspended computation indicates the type of its result. A separate process translates the graph into a particular programming language. Producing graphs rather than program text solves problems with code duplication and premature reduce/residualize decisions. Fuse's termination strategy, which employs online generalization, specializes conditional recursive function calls, and unfolds all other calls. This strategy is shown to be both powerful and safe.

1 Introduction

Program specialization (also called *partial evaluation*) transforms a program and a description of the *valid inputs* to the program into a *specialized program* that is optimized to work on those inputs. Program specializers employ a very simple transformational technique: the selective symbolic execution of the program. Expressions that can be reduced are reduced, while those that cannot be reduced appear in the specialized program. Program specialization has been investigated as an artificial intelligence tool [2, 17], and is a proven technique for creating compilers and compiler generators [20], rediscovering important algorithms [12], speeding up computations by two orders of magnitude [5], parallelizing scientific code [3], and optimizing programs [4]. Program specialization is less powerful than other transformational techniques such as the unfold/unfold transformation of [9] and the driving transformations of supercompilation [30]. Nonetheless, it is still a very powerful transformation technique. Unlike fold/unfold, it is automatic, and unlike supercompilation, it is rapid.

Research in the 1960's and 1970's stressed the optimizing properties of program specialization [2, 17, 21, 22]. Specializers were tools for optimizing programs. Research emphasized improving the quality of specialized programs, and operating on programs that were not written with the specializer in mind. Because the quality of a specialized program depends upon the number of reductions performed by the specializer,

*FUSE-MEMO 91-6. This paper will appear in the Proceedings of the Conference on Functional Programming Languages and Architectures, Springer Verlag, August, 1991. This research supported in part by NSF Contract No. MIP-8902764, and in part by Defense Advanced Research Projects Agency Contract No. N00014-87-K-0828. Erik Ruf is supported by an AT&T Bell Laboratories Ph.D. Scholarship.

much effort was expended to gather as much information as possible. Toward this goal, symbolic type systems, conditional contexts, and conditional side-effects were developed [17]. Relatively little research was done on automatic termination or automatic prevention of code duplication. The contributions of this period were both theoretical, *e.g.*, the development of the Futamura projections [14, 15], and practical, *e.g.*, the exhibition that program specialization is a powerful tool.

Research in the 1980's stressed self-application [19, 20, 8]. The contributions of this period were the development of automatic termination strategies, and of Binding Time Analysis (BTA), which allowed the construction of efficient self-applicable partial evaluators. The cost of achieving self-application and compiler generation was less optimization and generality, as it was discovered that self-application required a very simple *specialization kernel*. To keep the kernel small, it was reduced to making reduce/residualize/generalize decisions based upon binding time information, and not based upon actual data values. The type systems, conditional contexts, and side-effect handling of the 70's were abandoned. Another proven benefit of BTA based systems was speed of specialization. For example, the *action trees* of [13] provided a highly tuned method for performing specialization very rapidly.

Our goals match those of the 70's: strong optimization for a wide range of programs and programming styles. Rather than striving for self-application at the cost of accuracy, our goal is to make specializers that produce highly specialized programs, and do so for programs that are not designed with specialization in mind. In particular, we are interested in *online program specialization*, where the reduce/residualize/generalize decision is made during specialization using actual data values, rather than *offline program specialization*, where the reduce/residualize/generalize decision is made using binding time information. The major cost of achieving our goal is the speed of specialization: online methods will often be slower than offline methods.

Four major issues must be addressed when designing a program specializer.

Residual program structure: Specializers beta substitute the program text itself. When a formal parameter appears more than once, code may be duplicated. Code duplication expands code size, and can change the time complexity of an algorithm [28]. Obviously, a specializer should avoid duplicating code.

Termination: Specializers symbolically execute code and explore program paths that may not be explored at runtime. When a specializer explores an infinite program path that may not be explored at runtime, it will fail to terminate. (Failing to terminate if the program itself would fail to terminate is considered acceptable behavior.) The tension is between performing too little exploration, thereby producing a very poor specialization, and performing too much exploration, thereby risking divergence. Arbitrary cutoff methods usually produce poor specializations.

Information usage: There are many sources of information about data values that can be used during symbolic execution. Examples include information provided by the user when invoking a specializer; constants within the program being specialized; paths that lie on the true or false branches of `if` expressions, which allow the specializer to assume the truth or falsity of the `if` expression's predicate; and properties of the residual code created during specialization, such as the type or structure of the values the residual code will return at runtime. All specializers make use of the first two sources of information, while the other sources are used depending upon the ambition and goal of the specializer.

Polyvariance: A given program point may be specialized in many different ways, each using different information. The best specializations are produced when there is no *a priori* limit on the types or numbers of specializations that can be generated from a given program point. Some specializers only produce one specialization per program point (these are rare), some limit the specializations according to a given template (usually the offline specializers that employ BTA), and some set no limit (usually the online specializers). We will call a specializer that sets no limits on the types of specializations *completely polyvariant*.

The attraction of online program specialization is its simplicity and power. Structured values are as easy to reason about as scalars are, and higher order functions are only slightly more complicated. Online

specializers usually create completely polyvariant specializations. Because online specializers aren't self-applicable, they can have the ability to reason about residual code [31], employ conditional contexts [17], aggressively reuse specializations [26], and have a safe yet powerful termination strategy. Also, for structured objects constructed during specialization, the reduce/residualize decision can effectively be postponed until after specialization is complete, resulting in better specializations (c.f., Section 4).

The problem, until now, has been to create a fully automatic online specializer that produces highly specialized programs and solves the code duplication problem. Previous online partial evaluators either required user provided annotation to guide termination and code duplication [17, 27], or had weak or non-existent automatic termination methods [16, 21] and only worked on a small range of programs. In his (re-)examination of program specialization, Jones [18] described his concerns regarding the construction of an automatic online specializer. Jones was concerned that online methods would be too expensive, would not be general enough to be practical, would be very hard to make terminate, would not use nonlocal information, and would not be amenable to self-application. Except for not being amenable to self-application, the problems that he cites are all solved by this research. We show that the extra expense yields better specializations, that online methods can be made both general and practical, that termination can be accomplished, and that nonlocal information can be used.

This paper focuses on Fuse's solution to code duplication, and its termination method. To us, the problem of code duplication is not avoiding it, *per se*, but to do so without compromising the accuracy of the specializer. A specializer can't know with 100% accuracy what code is duplicated without first "duplicating" the code. Ideally, code duplication should effectively be ignored during specialization to allow highly specialized programs to be constructed. Fuse achieves this ideal through its use of *graphs.* Specialized programs are represented as graphs, not as program text. Computations are represented by nodes, and the flow of values by directed edges. A value that is needed in many places in the residual program will appear as a node with multiple fanout. Such nodes represent values that need to be bound with a `let` expression at code generation time. During specialization, reductions are made regardless of code duplication or the specialized program's eventual structure. Scalar and structured values (cons cells and closures) are handled identically. The code generator also reconstructs the necessary lexical structure to express the block structure possessed by graphs.

The termination properties of an automatic specializer lie in its handling of function calls [28]. It makes a *reduce/residualize/generalize* decision when applying a function: whether to reduce the call, *i.e.*, invoke the function on its arguments, or to create a *residual call* to a specialized function. When it chooses to produce a specialized function a *generalize* decision must be made: how much of the known information to use when creating the specialization. Choosing to reduce too few calls, or to use too little information, results in trivial specialization, whereas reducing too many calls, or using too much information, often leads to nontermination. The task is to get both choices right.

Fuse's termination strategy is based upon specializing closures that are used recursively, and upon online generalization. Fuse specializes the application of a closure when the application represents a recursive call that will be conditionally executed at runtime. Fuse specializes a closure on the generalization of the current arguments and the arguments from a pending (active) call, thereby building a specialization that can be used for the current and pending call. This strategy is very safe, but sometimes prone to premature termination. Special generalization rules are used lessen the amount of generalization performed, thereby yielding better specializations without risking divergence.

This paper has six sections. Section 2 presents an overview of the structure of Fuse. It discusses *symbolic values*, basic computational elements of Fuse. Fuse's termination strategy is given in Section 3. The benefit of graphs, and generating code from them, is presented in Section 4. Related research is discussed in Section 5. We conclude with Section 6.

2 Overview of Fuse

Fuse is designed primarily as an interpreter and secondarily as a program transformer. The specialized program is generated almost as an afterthought of interpretation. Functions are applied to *symbolic values* and return symbolic values. The structure of Fuse is very similar to an interpreter for an untyped functional language [1]. The major difference is the handling of `if` expressions and `call` expressions (applications). The change for `if` expressions is very minor, whereas the change handling for `call` expressions is very large, because termination decisions are made here. We will first discuss symbolic values, then discuss the structure of Fuse.

2.1 Symbolic Values

Symbolic values represent both the type of object that will be produced at run time and the code to produce the value. The result of interpreting (specializing) 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*, the value domain of the language whose programs are specialized. Besides representing a subset of *VAL*, the value attribute is also a value that is operated upon. For example, during partial evaluation the `car` operation takes the car of the value attribute of the symbolic value it is applied to. 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. Otherwise, it is called an *unknown value*. Value attributes range over the Scheme values supported by Fuse, and over the special markers `a-value`, `a-list`, `a-natural`, `a-boolean`, and `a-function`. (In the implementation these markers are represented by tagged values, not as Lisp symbols. For simplicity of exposition, this technical distinction is not made in the rest of this paper.) These markers are interpreted as representing all values in the type. The marker `a-value` represents all values. Any symbolic value whose value attribute is `a-value` is called *completely unknown* or *completely unconstrained*. Because symbolic values are first class objects, constructing structured symbolic values, such as the value that represents all pairs whose first element is a boolean and whose second element is a natural, is simple.

(Our notions of "known" and "unknown" differ significantly from the BTA terms "static" and "dynamic". "Static" and "dynamic" refer to expressions, such as identifiers and `if` expressions, not to values. An expression is *static* if all the specialization time values it denotes are *known*. An expression is *dynamic* when it may denote an *unknown* value during specialization. BTA terminology uses the term *partially static structure* to refer to an expression that always returns a structured value where certain slots will be known during specialization. We have no special terminology for a structured value that contains unknowns, we simply call them structured values.)

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. Symbolic values are very similar to IBL's *partial values* [27], except that the code attribute of a partial value is program text, whereas ours is a *graph structure*. The code specifying the creation of a value V only appears in the specialized program when V itself must be constructed at runtime. Code is either (1) an identifier, (2) an `if` expression, (3) a `call` expression, or (4) a value. When a symbolic value represents a known value, the code attribute is the value attribute. The two major benefits of using symbolic values are propagation of type information, and postponement of many reduce/residualize choices until code generation time. We will write a symbolic value using angle brackets, for example, a symbolic value with value attribute v and code attribute c will be written ⟨v,c⟩. A more detailed description of symbolic values appears in [31].

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. Cons pairs are written `(a . b)`.

```
(define peval
  (lambda (exp env stack)
    (cond ((constant? exp) exp)
          ((var? exp) (lookup exp env))
          ((if? exp)
           (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-code 'if (list pred then-sv else-sv))))))))
          ((lambda? exp) (create-closure exp env))
          ((call? exp) (papply (peval (call-head exp) env stack)
                               (peval (call-arg exp) env stack)
                               stack))
          ((letrec? exp) (peval (letrec-body exp)
                                (recursive-extend-env (letrec-bindings exp) env)
                                stack)))))
```

Figure 1: 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 closures.

⟨(⟨`a-boolean,(car (f a))`⟩ . ⟨`a-natural,(cdr (f a))`⟩),`(f a)`⟩. A pair to be created by a runtime call to `f`.

Fuse is invoked on a program and symbolic values. 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`. The symbolic values so constructed have empty code slots. Before starting symbolic execution Fuse *instantiates* the symbolic values, initializing their code slots. 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. (Actually, the code slot is set to take the `tc-car` and `tc-cdr` of the parent pair. The `tc-` functions are described later.)

2.2 Fuse's Structure

Fuse is structured as an interpreter for an untyped functional language [1]. It differs from a standard interpreter in it handling of if expressions and call expressions (Figure 1), and the operations of its primitive functions.

A syntactic preprocessor transforms input programs into a semantically equivalent program that uses just six different expression types. (See [29] for a motivation and implementation of using a small number of expression types for reasoning about programs.) The six primitive expression types are: `Constant`, `Variable`, `If`, `Letrec`, `Call`, and `Lambda`. Four of the six, `Constant`, `Variable`, `Letrec`, and `Lambda`, simply execute as normal. `Lambda` builds closures that contain an extra slot for caching the specializations of the closure. For `If` expressions, the predicate is partially evaluated. If it evaluates to either *true* or *false*, then the consequent or alternate is specialized. If the predicate evaluates to an unknown value, then a new symbolic value is created whose value attribute is `a-value` and whose code slot contains a residual `if` expression. `Call` expressions are the subject of Section 3.

The primitive functions, such as `+`, `cons`, and `cdr` operate on symbolic values and produce symbolic values. Known symbolic values produce known symbolic values, while unknown symbolic 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. The `tc-` functions know that their arguments have been typechecked, and do not perform redundant type checking.

The function `cons` always constructs a cons pair. When known values are paired together, the code slot contains the pair, otherwise it contains an instruction 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. The functions `car` and `cdr` only produce new code when handed `a-value` or `a-list`. The other structured value constructors, `vector` and `lambda`, also always construct objects during specialization.

3 Termination

Termination in all automatic program specializers is based upon some form of loop detection. At selected "program points," a cache is kept of the specializations made from the program point indexed by the states that produced the different specializations. Whenever the program point is to be specialized rather than unfolded, the cache is checked. If there is a hit, the cache entry is used as the specialization, otherwise, a new specialization is made and entered in the cache. Unless arbitrary bounds are used, automatic specialization only terminates when all loops that arise during partial evaluation are either self-terminating or broken by a cache hit.

The element of specialization (program point) in Fuse is the closure. Specializing a closure produces a *specialized closure* that contains three elements: a new name, the arguments that produced the specialized closure, and a symbolic value that contains the type returned by the specialized closure and the code of the specialized closure. We refer to the code of a specialized closure as a graph, and draw it as such (c.f., Section 4).

Fuse detects loops when detected when applying closures. For each closure, a cache is maintained that contains the specializations of the closure *indexed* by the types of the arguments that generated the specializations. Each time a closure is applied, its cache is checked; if the arguments have the same type as an index, then, instead of applying the closure, a residual call to the specialized closure is created and returned. (To maximize sharing of specializations, Fuse also maintains information that allows a specialization to be reused, without any information loss, when the arguments don't exactly match the cache index [26].)

Ensuring cache hits requires choosing to residualize certain calls to create cache entries of specialized closures, and abstracting the arguments to ensure a finite number of cache entries. When not enough calls are residualized, the specializer loops endlessly for lack of hitting in the cache; when the abstraction doesn't discard enough information, the specializer builds an infinite number of specializations. For example, consider the "counting down" implementation of the factorial function:

```
(define (fact n)
  (letrec ((loop (lambda (n)
                   (if (= n 0) 1 (* n (loop (- n 1)))))))
    (loop n)))
```

When specialized on an unknown number, the call sequence looks like

```
(fact <a-natural, n>)
(loop <a-natural, n>)
(loop <a-natural, (- n 1)>)
(loop <a-natural, (- (- n 1) 1)>)
...
```

If the none of the calls to `loop` are residualized, the specializer will loop. Residualizing one of the calls will effect termination because hits are based on type information. Before a closure is specialized, an entry is made in the specialization cache to hold the specialization that will be produced. When the recursive call is encountered while specializing the closure, a cache hit will occur, terminating the loop.

It would seem that residualizing all calls to closures would solve the problem of deciding which ones to specialize. Unfortunately, no information is gleaned from a residual call expression,[1] so information would be lost. The problem could be avoided somewhat by carefully writing code to be partially evaluated, but this would go against our goal of specializing as large a class of programs as possible.

Sometimes the arguments that a closure is specialized on must be abstracted before specialization. Otherwise, an infinite number of specializations will be made. Consider, for example, the "counting up" implementation of the factorial function:

```
(define (fact n)
  (letrec ((loop (lambda (m ans)
                   (if (> m n)
                       ans
                       (loop (1+ m) (* m ans))))))
    (loop 1 1)))
```

When "counting up" factorial is specialized on an unknown number, the calls to `loop` appear as

```
(fact <a-natural, n>)
(loop <1,1> <1,1>)
(loop <2,2> <1,1>)
(loop <3,3> <2,2>)
(loop <4,4> <6,6>)
(loop <5,5> <24,24>)
...
```

Clearly, deciding to residualize any given call, or all of them, would not ensure termination. In this example, termination is not an issue of unfold versus residualize, and specializers that rely only on unfold or residualize strategies will fail to terminate. To ensure termination, the arguments must be abstracted. Abstraction maps a value to a value that represents more values. For example, an abstraction of `1` is `a-natural`, and an abstraction of `a-natural` is `a-value`. In this example, abstracting both arguments to `a-natural` before specializing would achieve termination.

Fuse's primary termination strategy unfolds a recursive call (applies a closure) only when there is no match in the cache, and when it can prove that if the program made the previous call then it would also make the current call (Figure 2). To do so, it maintains a stack containing all active calls to closures, and the arguments the closures were applied to. The stack also contains *conditional markers*. The specializer places them on the stack when specializing the arms of `if` expressions whose predicates did not evaluate to either true or false. These markers indicate that the specializer has entered a *speculative region* of the computation: the program at runtime may or may not perform the computations being specialized. Before a closure is applied, the cache is checked. If the arguments match the index of specialized closure S, then a residual call to S is created. Otherwise, the stack is examined starting from the most recent entry. If there is no active call to the same closure, or there is no conditional marker before the most recent call to the same closure, then the call is unfolded. If there is a conditional marker before the most recent call to the same closure, it means that the call about to be made is speculative, and that unfolding it may lead to specializer loops that don't correspond to runtime loops.

When such a call is detected, Fuse elects to specialize rather than unfold. Abstraction of arguments occurs at this point, because merely specializing the closure may lead to infinite numbers of specializations.

[1] [31] describes an extension to Fuse that allows type information to propagate out of residual expressions. Given such a feature, a specializer could residualize every call, and use a postpass to collapse trivial calls. Such a mechanism, however, extracts an additional specialization time cost, as would the collapsing postpass.

```
(define (apply-closure cl args stack)
  (cond ((lookup-specialization-cache cl args)
         (make-residual-call (lookup-specialization-cache cl args) args))
        ((specialize? cl args stack)
         (let ((<args' stack'> (generalize-argument cl args stack)))
          (if (lookup-specialization-cache cl args')
              (make-residual-call (lookup-specialization-cache cl args') args')
              (make-residual-call (specialize-closure cl args' stack') args))))
        (else (primitive-apply-closure cl args stack))))

(define (specialize-closure cl args stack)
  (let ((sc (make-specialized-closure
             (gen-name)
             arg
             "to be replaced")))
    (cache! (closure-spec-cache cl) args sc)
    (set-specialized-closure-body sc (primitive-apply-closure cl args stack))
    sc))

(define (make-residual-call sc args)
  (make-sval *a-value* (make-code sc args)))

(define (primitive-apply-closure cl args stack)
  (let ((<formals body env> cl))
   (peval body
          (extend-env formals args env)
          (cons (cons cl args) stack))))
```

Figure 2: Application of compound procedures. Termination and specialization decisions are made here.

Fuse attempts to perform the minimal amount of abstraction while still ensuring termination. It does so by comparing the arguments of the call being specialized against all arguments of all active calls to the same closure. It finds the invocation whose arguments are the most similar to those of the current call, and then *generalizes* these together to form a new set of arguments. If the new arguments don't hit in the cache, a new specialization is created for the generalized arguments.

Generalization maps two symbolic values *s1* and *s2* into a new symbolic value S representing (at least) all the objects that *s1* and *s2* represent. Fuse's termination strategy uses generalization to discard information about values to ensure that only a finite number of specializations are made. When either of *s1* or *s2* are scalars, S's value attribute is the least upper bound of *s1* and *s2*'s value attributes. When both *s1* and *s2* are pairs, their cars are generalized and their cdrs are generalized. The code slots are created via instantiation.

As as example, we will show how Fuse specializes the Iota program (named after the operator of the same name from APL):

```
(define (iota n)
  (letrec ((loop (lambda (m)
                   (if (= m n) '() (cons m (loop (1+ m)))))))
    (loop 0)))
```

The stack starts as

```
(iota <a-natural, n>)
```

and after the first call to `loop` is

```
(iota <a-natural, n>)
(loop <0,0>)
```

and before the second call to `loop` is

```
(iota <a-natural, n>)
(loop <0,0>)
*conditional-marker*
```

Therefore, Fuse discovers that the second call to *loop* is speculative. Since there is no cache entry that this call hits, Fuse creates a specialization based on the generalization of the current argument `<1, 1>` and the previous argument, `<0,0>`, yielding `<a-natural, m>`. `Loop` is now specialized on this value, and with a cache entry indexed by the type `a-natural` to hold the result of the specialization. Specialization proceeds with the stack

```
(iota <a-natural, n>)
(loop <a-natural, m>)
```

At the next call to `loop`, its argument is `<a-natural, (1+ m)>` whose type, `a-natural`, hits in the cache. A residual call to the specialized closure in the cache is created, the creation of the specialized closure is completed. The final code is (with a little cleaning of variable names):

```
(define (iota1 n)
  (if (= 0 n)
      '()
      (cons
        0
        (letrec ((loop1 (lambda (m)
                  (if (= m n)
                      '()
                      (cons m (loop1 (1+ m))))))))
         (loop1 1)))))
```

This strategy produces "surplus" code because loops aren't specialized until the recursive call is hit, at which time the loop has already been unrolled once. Fuse heuristically eliminates this extra preamble. Whenever the arguments of active call A on the stack are generalized against some other arguments, it means that call A will be a preamble to a specialization. Therefore, when the invocation A is finished, instead of returning its result, Fuse returns a residual call to the specialization that was created. Applying this heuristic to the example yields the code that Fuse produces:

```
(define (iota1 n)
  (letrec ((loop1 (lambda (m)
                    (if (= m n)
                        '()
                        (cons m (loop1 (1+ m)))))))
    (loop1 0)))
```

(Offline systems that use a monovariant BTA that does not duplicate code, and therefore assigns one BTA signature per function definition, do not produce preambles.)

Fuse's strategy is very safe. For example, it avoids divergence in the presence of *counters* (called "static values under dynamic control" in [18]), which are values that are always known, but that don't affect control flow. Speculative loops that perform a test before iterating are eventually halted through the generalization mechanism. Fuse will fail to terminate only when the program being specialized itself fails to terminate for any valid input, or when the program contains a non-terminating region regardless of input. However, in its eagerness to terminate, Fuse will sometimes terminate too early because it generalizes away too much information.

To the basic termination strategy Fuse adds rules that either allow unfolding where specialization had before been required, or that prevent generalization when specialization is chosen. In either case, the rule is not allowed to change the termination properties of Fuse: any program which Fuse would terminate on without the extra rules it must terminate on when new rules are used.

As an example of premature termination, consider compiling a program P written in language L by specializing an interpreter I for L on P. To ensure that the constituent elements of conditional expressions in P are compiled, the arms of the conditional statement within I must be specialized even though the predicate of the conditional will not be known until runtime. More concretely, a the fragment of I that interprets conditional statements in L may appear as

```
(if (conditional-expression? exp)
    (if (eval (predicate exp) env)
        (eval (then-arm exp) env)
        (eval (else-arm exp) env))).
```

To fully compile the program, all calls to `eval` in this fragment must be unfolded (or specialized) without losing the value of `exp`. Unfortunately, the calls to `eval` in the consequent and alternative of the inner if are speculative, and generalization here loses the value of `exp.`

We have two approaches to solving this problem, one that is fairly standard, and one that is novel, but unimplemented. The standard approach is based upon detecting structural inductions in the program [28]: unfold instead of specialize when there is some bounded argument that gets smaller on each recursive call . At each subsequent recursive call, the same argument must get smaller for unfolding to be performed again.

The other method exploits the observation that we want the specializer to "consume" all of the inputs. The purpose of program specialization when compiling programs is to translate the entire source program into the language the interpreter is written in. Therefore, a method for achieving the goal of traversing all input structures is to remember which objects are part of the input (of which there are only a finite number), and which are created during specialization. When unfolding a speculative call, it is safe to unfold when patterns of input values are different from previous patterns of input values. Since there are only a finite number of input values, this strategy will not cause the partial evaluator terminate any less often than before. Considering the interpreter fragment above, `eval` will be unfolded as long as different pieces of the

input appear in `exp`. Only when the same value reappears in `exp` will a specialization occur, at which point the environment will be generalized.

The termination strategy is adequate for programs which, when specialized, produce a graph without residual unspecialized closures. However, some programs produce higher-order residual code; in this case, unspecialized closure objects will appear in the final graph whenever a closure is an argument in a residual function call. Because of Fuse's design, these closures must be specialized before the code generator that turns graphs into target code can run. When specialization is complete, a postpass finds each residual unspecialized closure and specializes it on completely unknown inputs. (A similar technique is proposed in [24].) It would be more accurate to specialize each closure on an approximation to the arguments of all its potential call sites, but we haven't yet implemented the pass to collect this information.

Stream-like objects, such as the Y operator, cause a minor problem because specializing a closure may result in a new residual closure, whose specialization produces a new residual closure, etc. For example, consider

```
(define (evolve-system f s)
  (let loop ((s s))
    (cons s (lambda () (loop (f s))))))
```

Specializing this function on unknown values produces a graph with a residual closure. Specializing that closure will yield yet another closure.

The termination strategy described above fails in this case, because it relies on the presence of a conditional expression with unknown test to signal entry into a speculative region, thus pushing a conditional marker and forcing generalization. In this case, specializing the residual closure is speculative, not because the specializer was unable to decide a conditional, but because it cannot prove that the residual closure will indeed be applied at runtime.

At the moment, Fuse relies on a heuristic which forces the pushing of a conditional marker when specializing a residual closure. Using this heuristic, specializing `evolve-system` on `1+` and 0 will produce

```
(define (evolve-system1 f s)
  (cons 0 (let loop ((s 1))
            (cons s (lambda () (loop (1+ s)))))))
```

There is a preamble before the loop, as usual, but the proper loops are formed. This mechanism also properly handles the Y operator.

This solution is not ideal; although it works in many common cases, such as that shown above, it is sometimes overly conservative, and fails to terminate in some pathological cases. We are actively researching a better strategy. Other specializers that handle higher order programs [11, 7] avoid this termination problem through the use of monovariant BTAs.

4 Graphs

Using graphs to represent code in a program specializer was pioneered by Berlin [3, 5]. This technique allows program structure to be ignored during specialization without risking code duplication. A postpass constructs executable code from the graph, at which time the proper binding constructs are inserted into the generated code. The important differences between our graphs and Berlin's graph are that our graphs may be cyclic so that they can express loops, and may themselves contain subgraphs so that they can express block structure.

A node in a graph (Figure 3) is either a primitive function, an application node (written as `@`), an `if` node, or a `lambda` node, which corresponds to a specialized closure. Our graphical representation solves code duplication problems, as well as allows the delaying of some reduce/residualize decisions until after specialization is complete. The cost of this added power is the cost of generating executable code from the graph. If we assume that the specialized program is compiled before being executed, then this additional

cost is relatively small, because the work performed by our code generator is a subset of the work that a compiler performs. The major algorithm run by the code generator is computing dominators, a standard compiler task.

4.1 Code Duplication

During conventional program specialization there is a substitution of formal parameters by actual parameters. Some of the actual parameters will be represented by the code that computes them. If these expressions are substituted wherever formal parameters occur, then code will be duplicated when a formal parameter appears more than once. This can change both a program's complexity [28], and, in the presence of side-effects, its meaning.

However, in order to achieve the best specialization, it is often (especially in scientific code [5]), necessary to perform reductions that risk such duplication. For example, consider the function for adding two complex numbers represented in rectangular coordinates:

```
(define (+complex x y)
  (make-complex (+ (real x) (real y))
                (+ (imag x) (imag y))))
```

We would like the specialization of

```
(define (+complex3 x y z)
   (+complex x (+complex y z)))
```

to produce

```
(lambda (x y z)
  (make-complex (+ (real x) (real y) (real z))
                (+ (imag x) (imag y) (imag z))))
```

This result can only be obtained if multiple substitutions are allowed, because the original definition of `+complex` has multiple occurrences of both `x` and `y`. But allowing such substitutions may lead to code duplication, because one cannot tell if there is duplicated code until after all substitutions and simplifications have been performed.

Because Fuse builds graphs, and produces program text in a separate pass, it avoids these problems. Graphs express shared values, so that the control part of the specializer need not worry about sharing or duplication. The code generator introduces the necessary binding constructs (`let` expressions) to avoid code duplication.

Specializers that operate by pure beta-reduction either can't reproduce this result because they outlaw duplication on the same control thread (e.g., [8]), or can reproduce this result but run the risk of really being left with duplicate code (e.g., [28], which allows some duplication in carefully controlled instances).

4.2 Delaying the Reduce/Residualize Decision

One major benefit of symbolic values is that because structured objects are built at specialization time, with the code to construct them attached to them, only when specialization is complete is the decision to produce residual code made. If a value is fully consumed during specialization, it will not appear in the output graph, and the code to construct it need not be issued. For example, consider the following program fragment.

```
(let ((local-f (lambda (x y) (+ (* x x) (* y y)))))
  (if (pred42 z)
      (local-f z (g z))
      (list 1+ local-f)))
```

The question is whether to leave the `let` binding residual or to reduce it. A specializer cannot make the correct choice until after the evaluation of `pred42`. When `(pred42 y)` evaluates to *true* then reducing the binding allows the function to be run and discarded at partial evaluation time. Otherwise the lambda expression must appear in the specialized program. Symbolic values solve this problem, because they denote both the closure and the code to build it.

This example can be repeated for other types of structured values, such as pairs and vectors. In general, the specializer doesn't need to be aware of the reduce/residualize decision for structured values, as the symbolic value will represent both the object and the code to construct it. (Schooler's *, partial values* [27] also had this property.)

Most importantly, a structured value can be both used and left residual. For example, consider representing conditionals in the lambda calculus, where *true* is $\lambda xy.x$, *false* is $\lambda xy.y$ and a conditional is $\lambda pca.pca$. When there are residual conditionals left in specialized code, the `lambda` expressions representing *true* and *false* must also remain residual (unless transformations not usually performed by a specializer are employed). If having the *true* and *false* lambda expressions be residual prevents any reduction of *true* or *false*, no conditionals can be reduced, resulting in very poor specializations. Consider, for example, the following code.

```
(lambda (x y z)
  (let ((true (lambda (x y) x))
        (false (lambda (x y) y))
        (fif (lambda p c a) (p c a)))
    (let ((f> (lambda (x y) (if (> x y) true false))))
      (+ (fif (f> x y) 3 4)
         (fif (f> y z) 3 9)))))
```

If this code were specialized for `x` being 10, `y` being 20, and `z` being unknown, we would want the specializer to reduce `(fif (f> x y) 3 4)` to 4. But it could only do this if it could both apply true and leave its construction residual. Using symbolic values and graphs allows this to happen in Fuse.

4.3 Generating Programs from Graphs

The main issues in generating code are translating the parallel meaning of graphs into serial code (*i.e.,* choosing orderings in which to do things) and rediscovering block structure and lexical scoping. Translating a parallel graph into serial code is difficult because order of evaluation must be reestablished: the dataflow IFs of graphs must be permuted into the control IFs of serial programming languages. This problem is compounded by large fanout and reconvergences of the possible execution paths and data values. The Fuse code generator currently produces both Scheme and C programs.

Graphs have several characteristics that distinguish them from the style of program representation of our target languages. Graphs show only the dataflow of a program. The control information needed to evaluate the program is implicit and, in general, not uniquely determined. As an illustration, consider the graph of Figure 3. For brevity we refer to the computation represented by the node rooted at $+_2$, for example, simply as "$+_2$". There is no indication as to which of $+_1$ and $+_3$ should be evaluated first. Indeed, if the target language allowed parallelism to be explicitly represented, there would be cases for which we would not wish to specify the relative ordering of the computations. Also, although it is possible (in this simple example, trivial) to deduce that $+_3$ must be evaluated before the IF node, this ordering is not indicated explicitly.

Conditional nodes of graphs differ from the conditional expressions of many conventional languages in that the consequent, for example, may need to be evaluated even if the predicate is false (because the consequent may have multiple fanout, ie, be used in more than once place). This allows more flexibility when constructing the target program, at the cost of deciding for each node not only *where* but also *if* it is to be evaluated. We can introduce memoization (delay and force constructs) into the target program to allow more precise control over the computation.

Although graphs contain procedures (specialized closures), there is no explicit indication as to which computations are performed when a particular closure is applied. In the illustration we can see that, whereas

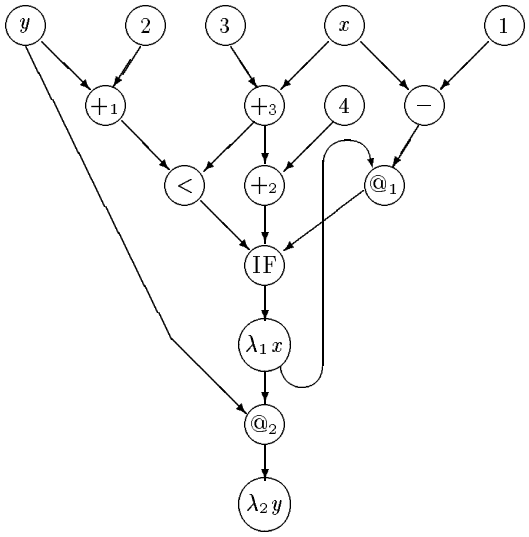


Figure 3: A Fuse graph. The λ nodes represent procedures (specialized closures). The node labeled $\lambda_1 x$, for example, represents a procedure λ_1 with a single formal parameter x, whose body is represented by the IF node. The @ nodes represent procedure applications. Nodes labeled + represent generic primitive operators.

$+_3$ must be evaluated on each application of λ_1, multiple applications of λ_1 for which the value y does not change can make use of a single evaluation of $+_1$. Performing computation outside the body of a closure in this manner is analogous to hoisting code out of loops as done by standard optimizing compilers.

We named the nodes in the example graphs for reference only; these names have no other significance. Names are generally used in conventional languages to allow values to be used repeatedly. These languages must therefore provide a rule, such as lexical scoping, for determining the value corresponding to a particular use of a name. Graphs use edges with arbitrarily large fanout rather than names to indicate such dependencies between nodes, so there is no need to represent within a graph any lexical structure or scoping.

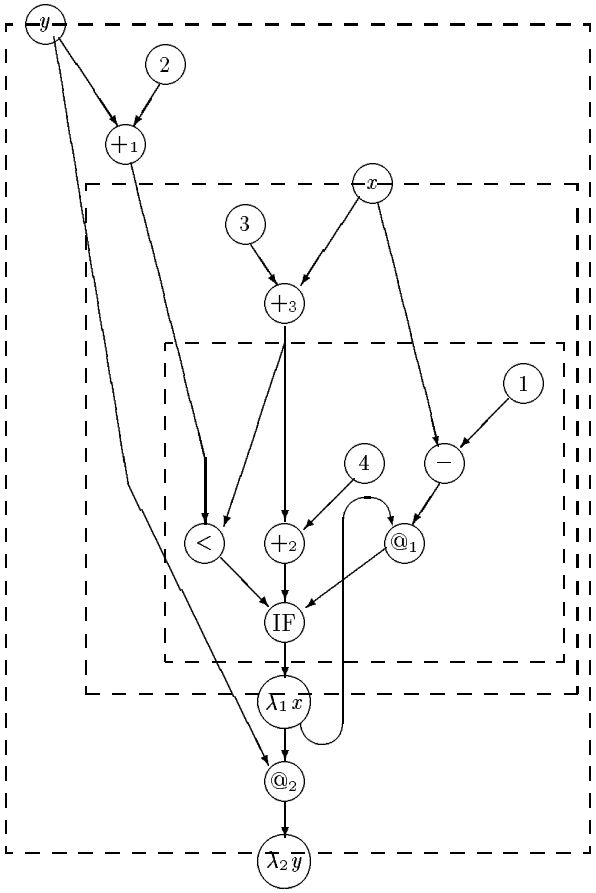
The essence of code generation is making explicit the information left implicit or unspecified by a graph. As an illustration of how such information might be represented, examine the *augmented graph* of Figure 4. This is the same graph as appeared in Figure 3, with boxes overlayed to indicate a block-like structuring of the computation. The box passing through each λ node delineates the entirety of that procedure. We can now see that $+_1$ is not a part of the body of λ_1, and that its evaluation should precede that of $+_3$ (which *is* a part of the body of λ_1).

Each box corresponds to some computation, with values crossing into the box corresponding to the free variables of that computation. The two edges leading from $+_3$ into the inner box have been coalesced into a single edge that forks at the box's boundary. Such an edge indicates a value that will be assigned to a variable in the target program, and the point at which it crosses into a box corresponds to the point where its value will be named. This, along with the tree-like nesting of the boxes, forms a lexical structure that aids deriving the lifetimes and scopes of the values in the target program.

Finally, the tree-like structure within each box allows for a straight-forward reading of the sequence of those computations. The correspondence between the augmented graph and the Scheme code in Figure 4 should be clear. The main task of code generation is to (effectively) produce augmented graphs.

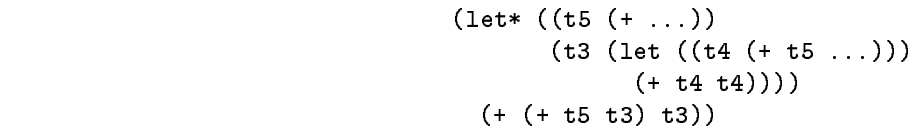
4.3.1 The Def-Point Tree

To impose structure on a graph, we introduce the *def-point tree*. The def-point tree of a graph is the skeleton of the structure of its target program. It is essentially a representation of the information found in the




```
(lambda (y)
  (let ((t1 (+ y 2)))
    (letrec ((p1 (lambda (x)
                   (let ((t3 (+ 3 x)))
                     (if (< t1 t3)
                         (+ t3 4)
                         (p1 (- x 1)))))))
      (p1 y))))
```

Figure 4: The augmented counterpart to Figure 3's graph, and the corresponding Scheme code.



```
(let* ((t5 (+ ...))
       (t3 (let ((t4 (+ t5 ...)))
             (+ t4 t4))))
  (+ (+ t5 t3) t3))
```

Figure 5: A graph fragment, its def-point tree, and the generated Scheme code.

augmented graph described above. Its nodes are exactly the nodes of the graph, and its root is the root node of the graph. A node's parent in this tree is that node's *def-point*, short for *definition point*. The target program is structured so as to ensure that each node is *defined*—that is, the code for computing the node's value is evaluated—before or as part of the definition of its def-point.

The sharing of nodes between a graph and its corresponding def-point tree is a potential source of ambiguity when referring to these structures. To avoid confusion, we adopt the following terminology. The terms *producer* and *consumer* specify relations within a graph. A node is a consumer of each of the nodes on whose value it directly depends, and is a producer for each of the nodes that directly depend on it. In the graph representing the summation of nodes X and Y, for example, the + node consumes both X and Y. The terms *parent* and *child* specify relations within the def-point tree, with *ancestor* and *descendant* being their respective transitive closures. Children of the same parent are *siblings*. A node is said to be *shallower* than any of its proper descendants, and *deeper* than any of its proper ancestors.

As an illustration of def-point trees, consider the graph fragment of Figure 5. The figure also shows a fragment of the corresponding def-point tree. There are two types of edges. The single edge between $+_1$ and $+_2$ indicates that $+_1$ is the only consumer of $+_2$'s value and that $+_2$ is not being hoisted (more on hoisting later). In this case the code for $+_1$ will include the code for $+_2$.

A double edge indicates that the parent must be defined in an environment in which the child has already been defined. Thus the figure indicates that the definition of $+_4$ precedes that of $+_3$, which in turn precedes that of $+_1$. The ordering in which siblings are defined is also significant. The definition of $+_5$, for example, must precede all of the others, and in particular it must precede that of $+_3$, a consumer of one of $+_5$'s consumers. Rather than cluttering up our illustrations by drawing these ordering relations, we instead choose to draw the tree such that it is safe to define the siblings from left to right. The right side of Figure 5 shows the Scheme code that would be generated from this tree.

4.3.2 Choices in Building the Def-Point Tree

A unique choice of def-point for a particular node may not exist, but a single def-point must be assigned. Several, sometimes conflicting, factors guide the selection process. It is desirable to keep environments as small as possible and to minimize the live ranges of the values manipulated by the target program. For example, the intermediate results of a computation should often be used as soon as they have been

determined. This argues for placing some nodes deep in the tree. On the other hand, code hoisting out of loops is achieved through the shallow placement of some nodes.

The placement of nodes in the def-point tree affects the termination properties of the resulting target program. We choose not to require that all computations performed by the unspecialized program be performed by the specialized one. Thus the specialized program may terminate more often than the unspecialized one. Since, in general, evaluating a node may be expensive or even nonterminating, we ensure that no computation is performed by the specialized program unless it is likewise performed by the unspecialized one.

A conditional node is said to *guard* nodes that may be evaluated on one arm of the conditional but not the other. Such a conditional is referred to as a *guard*. A node must be deeper than any of its guards. Otherwise, we say that a guard has been *violated*, possibly allowing the evaluation of a node not evaluated in the unspecialized program. The prohibition against violating the guards of conditionally executed code prevents the specialized program from executing code that would not have been executed by the corresponding unspecialized program. If a node appears only in the consequent of a conditional node, for example, it may not be shallower than the conditional node.

This restriction has two unfortunate consequences. Expressions that are invariant across multiple calls to a procedure may be evaluated repeatedly, and opportunities for the specialized program to avoid needless computations performed by the corresponding unspecialized program may be lost. The code generator safely hoists conditional code by employing memoization. Unfortunately, a description of the mechanism by which it detects and hoists invariant conditional code is beond the scope of this paper.

4.3.3 Assigning Def-Points

The algorithm for finding the def-points of the nodes in a graph begins with a graph traversal starting at the root node. The root is unique in having no parents, and it therefore has no real def-point. To get the recursive algorithm rolling, we assign the root node a fictitious def-point, and make it the first member and root of the def-point tree that we construct as the algorithm progresses.

Once a node's parents have all been added to the def-point tree, the def-point of the node itself can be found. First a bit of terminology. We can think of the def-point tree as being partitioned into procedures, with node N belonging to procedure P if P is the first procedure node on the direct path from N's parent to the root. A node is *definable* within a procedure if each free variable on which the node depends is a formal parameter of some procedure node on the direct path from the node to the root. Now we can formulate a preliminary version of the rule for determining a node's def-point. Recall that we must ensure that a node is defined before it is used, and that no computation is performed by the target program unless it would also be performed by the source program. In addition, we wish to minimize the lifetimes of the intermediate results of computations, and we wish to hoist code out of loops where possible. For the def-point of a node, therefore, we choose the deepest common ancestor within the shallowest procedure for which the node is definable, and for which no guards are violated. The phrase "common ancestor" refers to an ancestor of all of the node's parents.

The above rule is not sufficient, however, since it does not properly manage cycles in the graph. Progress is stalled when each of the nodes that has not yet been added to the def-point tree has a parent that likewise has not yet been added. We account for this by having the specializer keep track of procedure-level lexical structure as it builds the graph. This information is used to break the cycles.

5 Related Research

5.1 Related Automatic Termination Research

The other realistic research on automatic termination methods has been for offline specializers. These methods fall into two classes, which we shall call the Mix [28] approach and the Similix [8] approach. Both methods perform a Binding Time Analysis (BTA) to determine binding time information, ie, which expressions will be dynamic during specialization and which will be static. Mix employs a pass that uses the

binding time information to annotate each call with an instruction to the specializer whether to unfold or specialize the call. This pass considers both termination and code duplication when annotating the program. Similix hoists `if` expressions whose predicates are dynamic (called *dynamic ifs*) into their own procedures, and then always specializes these and only these procedures. Both methods usually terminate equally often, but the specializations produced by Similix can be better because Similix generally performs more unfolding. We view Similix as the more powerful method. Similix, like Fuse, builds residual specializations for only the speculative portions of the computation.

Where the offline termination strategies differ most from Fuse's strategy is in the lack of online generalization. Both Mix and Similix overload binding time information to play the role of an abstraction function. When residualization/specialization is chosen, any variable whose binding time value is *dynamic* is effectively boosted to `a-value` before specialization occurs, regardless of its actual value. Specializers that use binding time information as the abstraction function when producing specializations of program points fail to terminate on *counters*, values that are always known and change, but don't affect control flow. For example, we know of no offline specializer that terminates on the "counting up" factorial program or the `iota` program when given an unknown input. They correctly choose to residualize, but then produce an infinite number of specializations. Jones [18] proposed a method for performing further lifting of binding time values to solve this problem, but, there seems to be no published results regarding its use, nor has it been extended along with BTA to handle partially static structures and first class functions.

Although Fuse terminates more often than the offline methods, it sometimes prematurely terminates. We believe that with the additional structual unfolding rule, Fuse explores as much of a computation as Mix does while terminating more often. There are examples where Similix explores more of a computation than Fuse does and still terminates. This is to be expected: since Similix is willing to terminate less often, it can be more aggressive. The major difference between Fuse and Similix is that, when infinite specializations won't arise, Fuse does not know this and prematurely terminates, whereas Similix doesn't prematurely terminate. However, when infinite specializations do arise, Fuse stops, but Similix doesn't. Which strategy is superior depends on the specialization task at hand. We are investigating much more powerful termination strategies that would allow Fuse to continue unfolding in many more instances than it does now. These strategies, incorporated into an experimental specializer called *Melt*, would allow Fuse to dramatically increase the depth of exploration it could do without sacrificing termination.

5.2 Comparing Symbolic Values with *Partial Values* and *Q-tuples*

Symbolic values are closely related to IBL's *partial values* [27] and REDFUN-2's *q-tuples* [17]. (We collectively refer to these objects as *type/code objects*.) All solve the problem of associating type information (as well as other information) with residual code. The differences involve the status of the type/code object itself, what an individual type/code object can represent, and the representation of residual code.

Fuse and IBL, which are interpreters, treat type/code objects as first class values: *i.e.*, they are members of data structures, arguments to functions, and results of functions. Describing structured objects with partially known slots is easy: one simply constructs the object (which is itself a type/code object) out of type/code objects. REDFUN-2, which is designed as part of a "Program Manipulation System," treats type/code objects as the result of a transformation, they are not values passed to a program (though the known values they represent can be passed to a program). In particular, they cannot be elements of data structures, which prevents REDFUN-2 from representing, say, a three element list whose first element is the symbol `a` and whose other elements are integers. Another drawback is that information is lost during simplification. Consider `(cons x y)` where is it known that `x` is either 7 or 8. This information is lost when the q-tuple representing the residual cons expression is constructed. If the car of the q-tuple is taken, the symbol `x` is produced, but it is no longer known that its value is either 7 or 8.

IBL's partial values are not as first class as Fuse's symbolic values. There is no method for the user to create them, or to direct specialization using them. This is because IBL is viewed as an adjunct to a compiler, not as a stand-alone specializer.

The type structure of q-tuples is richer in some ways than that of symbolic values or partial values. Al-

though q-tuples do not express structured values, they can represent arbitrary sets of values, which symbolic values and partial values do not do. For example, a q-tuple can represent the set $\{3, 4, 7\}$, or the set {`alice`, `mary`, `john`}. A q-tuple can also represent specific values that will never be produced at runtime. This richer type structure allows for more accurate specializations.

Both partial values and q-tuples represent code as a tree. This results in code duplication. For the experiments Schooler performed, this duplication was quite a problem. Because symbolic values represent code as graphs, code duplication is not a problem.

5.3 Graphs and Specialization

Perlin [25] uses what he terms *Call Graphs* as the target of partial evaluation. His system differs from FUSE in that it doesn't have typed values, the graphs it produces must correspond to straight-line programs (their control can't depend on the unknown inputs), call graphs cannot call each other directly, and the user controls how the graph is built. It doesn't appear as if intermediate data structures are eliminated. His system has the advantage of interpreting the graphs directly instead of translating them into program text. By remembering the results of computations, his system can incrementally evaluate programs under changing inputs, an important feature for aritificial intelligence production systems, which were Perlin's domain.

6 Conclusions and Future Research

We have solved our goal of constructing a fully automatic and powerful online program specializer. Fuse has excellent termination properties and produces highly specialized programs. We have replicated many of the published experiments in partial evaluation. In [31] we show Fuse compiling programs, and doing so for a naturally written interpreter. We are also now capable of reproducing the work reported in [5] completely automatically. That original work, which dramatically applied program specialization to compiling scientific code, was performed using a manually guided specializer.

Fuse scores well on the issues of residual program structure (code and call duplication) termination, information usage, and polyvariance. Symbolic values allow the propagation of type information, and a graph representation for code allows reductions to be made without concern for program structure. In FUSE, the only decisions made during specialization involve termination, not program representation. Only after the program is partially evaluated do considerations of program layout arise. Because of this separation, FUSE performs extremely thorough specialization. Fuse's termination strategy correctly handles counters, while allowing for completely polyvariant specializations. Our decision to separate termination decisions from residual program structure will continue to pay benefits as Fuse is extended to make use of more information sources, such as the predicates of conditional statements.

We claimed in the introduction that the simplicity of online partial evaluation was a major attraction. Our experiences in using and extending Fuse bears this out. For example, Fuse has no difficulty with, or much knowledge of structured values that contain unknowns. Indeed, somewhat surprisingly, Fuse is virtually the first fully automatic specializer to handle pairs (cons cells) that may themselves contain unknown values. Mogensen [24], who first investigated *partial statics,* apparently never hooked his BTA directly to a specializer (his experiments were performed using hand written call annotations [23], although Sestoft's automatic call annotator [28] could have been used), and Schism [10, 11] is not automatic. While Similix-2 [6] can encode pairs as closures, its monovariant BTA, which doesn't duplicate code at BTA time, makes this encoding method a very fragile mechanism: information can be very easily lost. Another instance of Fuse's power is, that in the presence of higher order functions, Fuse's complete polyvariance can yield much better specializations than Similix-2 or Schism [11] can. (And does so without using macros to manually duplicate code.) These comparisons do not indicate that Fuse is better than Similix-2 or Schism, only that there is a tradeoff between self-applicability (which entails offline strategies) and simple, general, online methods that are not easily amenable to self-application. The other specializers listed here can specialize themselves, while the same cannot be said for Fuse.

Fuse solves many of the problems in constructing an online specializer that Jones anticipated. In particular, we have shown that online specialization is extremely practical, can be made to terminate, uses nonlocal information when it generalizes arguments with previous calls, and is not too expensive.

We have reported on only the basic elements of Fuse. Fuse has several interesting extensions. First, it reasons about residual code to determine the types of values that residual expressions will produce at runtime [31]. This additional information produces much better specialization for a wide class of programs. Second, it avoids redundant specializations [26]. Redundant specialization occurs when two different sets of arguments produce the same specialization. This has not yet been a major problem because specializers have used very weak descriptions of partial objects. As the descriptive system gets more powerful, for example, by the inclusion of a type system such as Fuse's, redundant specialization becomes a more severe problem. Solving redundant specializations also ameliorates some of the difficulty with reasoning about residual code, which requires computing fixed points. The cost of computing these fixed points drops dramatically when redundant specializations are avoided.

We want Fuse to incorporate more of the Scheme programming language. Scheme includes object identity, side-effects, and continuations. We intend to make side-effects explicit in the graph. This means having all side-effects take the resources they mutate (*e.g.,* the store, ports, and identity markers) as arguments and return the mutated resources. This approach turns control flow constraints into dataflow constraints, and simplifies reasoning about side-effects.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[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*, 23(12):25–37, December 1990.

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

[7] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhangen, Copenhagen, Denmark, December 1990.

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

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

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

[11] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, 1990.

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

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

[14] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

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

[16] M. A. Guzowski. Towards developing a reflexive partial evaluator for an interesting subset of LISP. Master's thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, January 1988.

[17] 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.

[18] 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.

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

[20] 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.

[21] K. M. Kahn. A partial evaluator of Lisp written in Prolog. Technical Report 17, UPMAIL, Department of Computing Science, Uppsala University, Uppsala, Sweden, February 1983.

[22] L. A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In Berkeley and Bobrow, editors, *The Programming Language Lisp*, pages 204–219. MIT Press, Cambridge, MA, 1964.

[23] 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.

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

[25] M. Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 122–128, 1989.

[26] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, CN, June 1991.

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

[28] 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.

[29] G. L. Steele Jr. _Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1978.

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

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