[7] Bondorf, Anders, and Danvy, Olivier, "Automatic Autoprojection for Recursive Equations with Global Variables and Abstract Data Types," DIKU Report 90/04, University of Copenhagen, Denmark, 1990.

[8] Consel, Charles, "New insights into partial evaluation: the SCHISM experiment," *Proceedings of the European Symposium on Programming (ESOP) '88,* Nancy, France, LNCS 300, pps. 236-246, 1988.

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

[10] Guzowski, M. A., *Towards Developing a Reflexive Partial Evaluator for an Interesting Subset of LISP,* Masters Thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, January 1988.

[11] Jones, N. D., Sestoft, P., and Søndergaard, "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.

[12] 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.

[13] Perlin, Mark, "Call-Graph caching: Transforming programs into networks," in the *Proceedings of the 11th International Joint Conference on Artificial Intelligence,* pps. 122-128, 1989.

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

[15] Sestoft, Peter, "Automatic call unfolding in a partial evaluator," in [6], pps. 485-506, 1988.

[16] Steele, Guy L., *Rabbit: A Compiler for Scheme,* MIT Artificial Intelligence Laboratory Technical Report AI-TR-474, Cambridge, MA, 1978.

[17] Weise, Daniel, "Automatic call unfolding based on dynamic IFs and generalization," in preparation.

Perlin [13] also uses what he terms *Call Graphs* as the product of partial evaluation. His system differs from FUSE in that it doesn't have partial values, the graphs it produces must correspond to straightline programs (they can't depend in any way 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.

Our future research revolves around partial evaluation of the full Scheme programming language and inventing better strategies for handling termination. There has been some activity in partially evaluating Scheme programs [14, 8, 10, 7], but much research remains to be done before arbitrary Scheme programs, which contain object identity, side-effects, first-class functions, and continuations, can be can be successfully partial evaluated. We intend to handle these language features in different ways. For example, we will make side-effects be 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 data flow constraints and simplifies reasoning about side-effects. We believe that significant parts of a partial evaluator's work could be eliminated by a more intelligent function caching strategy, will investigate better caching strategies.

References

[1] Appel, Andrew W., and Jim, Trevor, "Continuation-passing, closure-passing Style," in the *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages,* pps. 293-302, 1989.

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

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

[4] 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.

[5] 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.

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

4.4 Solving the Three Problems

We claimed that graphs and partial values solved the problems we noticed with STS partial evaluators. We now return to these issues to back up our claims.

Code Duplication Code duplication ceases to be an issue because pointers instead of text are used. Values can get to the functions that operate on them without changing the meaning or complexity of the program.

Partial Type Information Partial Values have an explicit type field for declaring their types. Manifestly typed user structures also work because structures can be freely car'd and cdr'd even when some slots are unknown during partial evaluation.

Either/Or Reduce/Residualize In FUSE all computations, except for certain recursive calls, are simply run and return values. The code generator produces program text for the nodes (partial values) of the graph. Expressions can be both reduced and left residual, a very important feature for thorough partial evaluation.

5 Code Generation

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 data flow IF 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. We have a brute force exponential routine for choosing the orders in the operations take place. It has worked well in practice so far, largely because the pathological conditions that would break it do not seem to arise. If this fails then we will trade optimality for speed.

Recovering block structure and lexical scopes is a major problem. We currently use heuristics for recovering this information, but are about to shift to using *environment conversion* [1], which eliminates block structure, and would finesse the problem at no cost.

6 Conclusions, Related Work, and Future Research

We showed how using graphs as the intermediate representation during partial evaluation naturally avoids and solves several difficult problems. In FUSE the only decisions made during partial evaluation involve termination, and not program representation. Only after the program is partially evaluated do considerations of program layout arise. Because it uses graphs, FUSE performs more thorough partial evaluation than previously achieved. For example, the circuit simulator scenario we presented could not be run by conventional partial evaluators.

Figure 4: The graph produced by partially evaluating the `sqrt` function.

Figure 5: The graph produced by partially evaluating `(lambda (s) (add-fred-to-3 (swap-2-3 (put-mary-in-2 s))))`. The shaded nodes represent values that were fully consumed during partial evaluation and do not appear in the residual program, which is `(lambda (s) (list (first s) (third s) (list 'fred 'mary)))`.

4.3 Example: Building a Graph

For example, consider a method for finding roots of systems by the fixed point method:

```
(define (create-fixed-point-method guess-method improver good-enough?)
  (lambda (x)
    (define (loop x guess)
       (if (good-enough? x guess)
           guess
           (loop x (improver guess x))))
    (loop (guess-method x))))
```

This procedure encapsulates the notion of a fixed point process. It accepts procedures defining the startup method, the improvement method, and the convergence test. It returns a new procedure that calls these procedures. We can define the square root function via

```
(define (abs x) (lambda (x) (if (< x 0) (- 0 x) x)))
(define (average x y) (/ (+ x y) 2))
(define sqrt (create-fixed-point-method
               (lambda (x) 10)
               (lambda (x g) (average x (/ x g)))
               (lambda (x g) (< (abs (- x (* g g))) .01))).
```

the graph resulting from `(sf sqrt (make-partial-number))` (Figure 4) has many attributes. First, and most importantly, the only calls left are to the single specialized procedure. All of the other function calls were completely executed at partial evaluation time. Call nodes take as an argument the graph representing the function to be called. The other attribute is the data flow attribute: `if`s are now selectors of values rather than control constructs. This affect code generation, as we point out below.

We also present the graph that would be produced by partially evaluating `(lambda (s) (add-fred-to-3 (swap-2-3 (put-mary-in-2 s))))` on an unknown input (Figure 5). This graph clearly shows cons cells being created at runtime, and also being left residual. The shaded nodes represent cons cells that were fully consumed during partial evaluation and that do not need to be residual. The other cons cells do need to be residual, so code is produced for them, which is just `(lambda (s) (list (first s) (third s) (list 'fred 'mary)))`.

```
(define (extended-number? x)
  (or (number? x) (partial-number? x)))
(define (pe-+ x y)
  (cond ((and (number? x) (number? y)) (+ x y))
        ((and (extended-number? x) (extended-number? y))
         (create-partial-value 'number + (list x y)))
        ('else (error "bad types to +" x y))))
```

The difference between primitive and computed partial values arises when structured values, such as lists and vectors are considered. For example, there are two kinds of partial pairs, computed partial pairs, whose representation includes its car and cdr (although they may themselves be partial values), and primitive partial pairs, where the car and cdr are unknown. The `car` function treats these two objects separately: when operating on a primitive partial pair, it creates a new partial value that represents a runtime `car` operation; when operating on a computed partial pair, it simply returns the car without doing any further work.

A syntactic processor transforms input programs into a semantically equivalent program that uses just six different expression types. (See [16] 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`. Three of the six, `Constant`, `Variable`, and `Letrec`, simply execute as normal. `Lambda` builds procedures as usual, but the procedures it builds have an extra slot for caching the specializations of the procedure. For `if` expressions, the predicate is partially evaluated. If it evaluates to either *true* or *false*, then the consequent or alternate is partially evaluated. If the predicate evaluates to a partial value, then a new partial value is created whose procedure slot contains `IF`, and whose arguments are the partial evaluations of the predicate, consequent, and alternative.

The intelligence of an automatic partial evaluator lies in its handling of `call` expressions [15]. It decides whether to simply run the call (reduce it) or to leave the call residual (produce a partial value that represents making the call at runtime). Termination and accuracy of partial evaluation revolve around this decision. Choosing to reduce too few calls results in trivial specialization, whereas reducing too many calls often leads to nontermination. FUSE's methods for making this decision are outside the scope of the paper, interested readers are referred to [17]. When the decision is made to leave a call residual, the function being called is specialized for the arguments to the function, then a partial value representing the call is produced and returned as the result of partially evaluating the call. When a function is specialized for a given set of arguments, those arguments and the specialized function are remembered in the function's cache.

When a function is to be specialized for arguments A, the partial evaluator first checks the function's specialization cache to see if the needed specialization has already been performed. The arguments A are compared against the stored arguments in the cache. When the arguments match the arguments of a cached specialized function in all the known values, then the cached specialized function is used as the specialization of the call.

the computation specified by the program is simply run on real and partial values. Computations on real values simply happens, computations on partial values leave a trace in the form of a graph. The nodes of the graph represent both a computation waiting to happen at runtime and the result of a partial computation. The inputs of a node represent the inputs to the computation. A graph is built for each specialized procedure created during partial evaluation. This section discusses partial values, presents the structure of the partial evaluator, gives an example, and shows how graphs solve the problems listed in the previous section.

4.1 Partial Values

A partial value represents values that may occur at runtime. The more that is known about the partial value, the fewer possible value it represents, and the more evaluation that can be done at partial evaluation time. There are two classes of partial values: *primitive partial values* and *computed partial values.* Primitive partial values are first class objects created by the user to declare promises about the inputs a program will receive at runtime. The user can only create primitive partial values, they cannot be examined or operated upon in any way by the user. Computed partial values are created during partial evaluation to represent computations that will occur at runtime. The partial evaluator freely manipulates partial values.

Both kinds of partial values are represented by a structure consisting of a type code, the procedure to create the value at runtime, and the procedure's arguments. Primitive partial values have a special marker in the procedure slot. There are constructors for each type of partial value. For example, the function `make-unknown-partial-value` makes a new partial value that nothing is known about, other than its identity. The function `make-partial-number` creates a value that stands for a number, and `make-partial-pair` creates a value that represents a pair but where neither the car nor cdr are known.

Partial values are nodes in the graph that partial evaluation builds. Every time a new partial value is created the graph is extended implicitly extended because partial values point to the nodes (partial values) they use as inputs.

4.2 The Partial Evaluator

We discuss here the handling of primitive functions, primitive expression types (special forms), function specialization and caching, and the relationship of partial evaluation to machine learning.

The primitive functions, such as `+` and `car`, have a different behavior during partial evaluation, when they accept both real and partial values. When they receive real values, they compute as normal. However, when they receive partial values, they may return a new partial value, where the partial value's creation function is the function that was called, its arguments are the arguments of the call, and its type code is determined by the primitive.

For example, the behavior of `+` is coded as

Doing so saves the creation and garbage collection of 2 needless states, and constant folds the consing of the symbol `fred` onto the list `(mary)`. Unfortunately, STS partial evaluators do not create such an efficient routine because they do not form the intermediate steps for fear of duplicating code. For example, consider merging `put-mary-in-2` with `swap-2-3`. An intermediate step of the merge is

```
(list (first (list (first s) '(mary) (third s)))
      (third (list (first s) '(mary) (third s)))
      (second (list (first s) '(mary) (third s)))),
```

which clearly shows the code blowup that conventional technologies outlaw.

3.2 Textual expressions carry no partial information.

Source language expressions are not designed to convey partial information about the value they denote. For partial evaluation to be thorough, partial information about a return value is required. This is especially true in object-oriented systems where efficient type dispatching is vital. Even if a particular return result is not known, knowing its type or its structure is a boon. Type checking, as well as method lookup and dispatch can still occur when the type is known.

3.3 Reduce/Residualize Choice is Either/Or

Conventional partial evaluation technology chooses whether to reduce a given expression or leave the expression (or a specialized version of the expression) residual in the specialized program. This firm either/or behavior inhibits optimizations. 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. Because `local-f` appears twice, the temptation is to leave it residual. But if `(pred42 y)` partially evaluates to *true* then reducing the binding allows the function to be run and discarded at partial evaluation time. Often the decision of whether to reduce or residualize needs information that isn't known until further partial evaluation occurs.

4 Partial Evaluation Using Graphs and Partial Values

Our solution is to replace the syntactic, source-to-source view of partial evaluation, with a more computational view. We view partial evaluation as a special kind of interpretation,

3 Current Technology: Syntactic Rewrites

Conventional partial evaluation technology performs source-to-source transformations: it maps program text into program text. We call partial evaluators that use source-to-source transformations *STS partial evaluators.* This section presents the technical limitations of such partial evaluators. We have found three major problems with source-to-source methods. First, the need to prevent code duplication prevents arguments from reaching the functions that operate on them, thereby inhibiting further specialization. Second, syntactic expressions do not carry any information about the results denoted by an expression. Third, STS partial evaluators force an either/or decision regarding residualization versus reduction. We discuss these issues in turn.

3.1 Code Duplication

During conventional partial evaluation 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 [15], and, in the presence of side-effects, its meaning.

To avoid code duplication, STS partial evaluators do not substitute expressions when the formal parameter bound to the expression appears more than once.[1] Unfortunately, this restriction prevents further partial evaluation from occurring. Most importantly, the elimination of intermediate data structures that allowed the circuit simulator to speed up 90-fold is not performed by an STS partial evaluator.

As a simple example, consider a problem solver coded as an expert system. The state space might be a list three elements long, and the state transformers might include the following.

```
(def-state-transformer put-mary-in-2 (s)
  (list (first s) (list 'mary) (third s)))
(def-state-transformer add3-fred-to-3 (s)
  (list (first s) (second s) (cons 'fred (third s))))
(def-state-transformer swap-2-3 (s)
  (list (first s) (third s) (second s)))
```

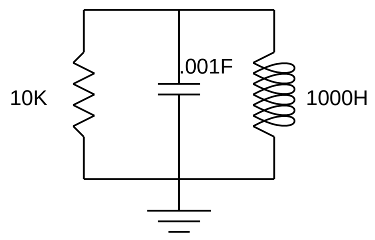
If the partial evaluator discovered an instance where the operators were always applied in the order `put-mary-in-2`, `swap-2-3`, and then `add-fred-to-3`, it would like to produce the specialized transformer

```
(def-state-transformer specialized-transformer-42 (s)
  (list (first s) (third s) (list 'fred 'mary))).
```

[1]Some partial evaluators have better substitution rules than others. For example, parameters that appear only once per control thread, ie, once per arm of an IF expression, can be substituted. While this substitution rule doesn't change the complexity or meaning a program, it can result in needless code blowup.

```
(LAMBDA (STATE)
  (LET*
      ((TEMP1 (VREF STATE 1))
       (TEMP3 (VREF TEMP1 1))
       (TEMP4 (VREF (VREF STATE 0) 0))
       (TEMP8 (VREF TEMP1 2))
       (TEMP12 (- (+ (* TEMP4 .02) TEMP8) (+ (* TEMP4 .00005) TEMP3)))
       (TEMP14 (* TEMP12 49.6277915633))
       )
    (VECTOR (VECTOR TEMP14)
            (VECTOR (* TEMP12 4.96277915633e-3)
                    (+ (* (+ TEMP14 TEMP4) .00005) TEMP3)
                    (- (* (- TEMP14 TEMP4) .02) TEMP8))
                    (+ .1 (VREF STATE 2)))))
```

Figure 3: Specialized program for the transient analysis of the simple RLC circuit. This function accepts a state at time t and returns the state at time $t + 0.1$.



The details of the simulator aren't important, what is important is that the simulator dynamically creates sparse matrices, sums them, and solves them. It also uses object method lookup and dispatch to create the matrices. The specialized program that the partial evaluator produced is striking for its simplicity, straight-lineness, compactness, and lack of structured values. No vestiges of the matrices produced or consumed during compilation, or of the control structures for doing so, or of the matrix inversion, or of the function retrievals and applications to implement object-oriented dispatching appear in the final code. Our experiments show the specialized simulator operating roughly 90 times faster than the unspecialized simulator. The use of graphs is crucial to these results. Without them, few of the matrix operations or object method dispatches could have been performed at partial evaluation time.

```
(define (next-state circuit state h)
  (let* ((matrix (create-integration-matrix circuit state h))
         (new-voltages (solve-matrix (trim-ground matrix)))
         (new-currents (compute-b-currents circuit new-voltages state h)))
    (create-new-state new-voltages new-currents (+ h (state-time state)))))

(define (create-integration-matrix circuit state h)
  (let ((voltages (state-voltages state)) (currents (state-currents state)))
    (let loop ((components (circuit-components circuit))
               (matrix (create-nxn+1-matrix (circuit-number-of-nodes circuit))))
      (if (null? components)
          matrix
          (loop (cdr components)
                ((2t-component-integration-method (car components))
                 matrix voltages currents h))))))
```

Figure 2: Scheme Code Fragments for Transient Analysis. `Next-state` accepts a state at time t and returns the state at time $t + h$. It computes the node voltages by creating and solving a sparse matrix, and then computes the branch currents. The function `create-integration-matrix` uses object oriented techniques to collect the conductance and current contributions of the reactive components into the admittance matrix.

Figure 1: Schematic diagram of FUSE. It accepts a program, and a description of the program's inputs expressed as *partial values.* The partial evaluator produces a graph which is translated into an executable program.

---

as `+` and `max`, are extended to handle partial values. When presented with real values, they compute as normal. When presented with partial values, they return a new partial value representing the computation must be performed at runtime. Special forms also operate differently during partial evaluation. For example, when an `if` expression's predicate evaluates to a partial value, both the consequent and alternative of the `if` are partially evaluated, and the partially evaluated predicate, consequent, and alternative are bound in a new partial value representing an `if` to be performed at runtime. A `call` expression is either simply run, or the function being called is itself partially evaluated to specialize it, and then a new partial value representing a call to the specialized function is returned. A separate program transforms the graph into executable code. We have developed programs that produce C code and Scheme code.

The outline of this paper is as follows. Section 2 gives a example of FUSE producing highly optimized programs for a numeric program. Section 3 itemizes the drawbacks of conventional partial evaluation technology. Section 4 presents partial values and graphs, as well as describes the partial evaluator itself. Section 5 indicates how code is generated from graphs. Section 6 draws conclusions, presents related work, and discusses future research.

2 Example of Partial Evaluation

As an example of the power of partial evaluation we present a fragment of a transient analysis circuit simulator (Figure 2) and the code produced for that fragment (Figure 2) when `(partially-evaluate next-state rlc-circuit <partial-state> 0.1)` is executed. The item `<partial-state>` represents an unknown state. `rlc-circuit` represents a description of the following RLC circuit.

1 Introduction

Partial evaluation transforms programs into equivalent, more efficient programs, called *residual* or *specialized programs*, by performing a symbolic execution that executes as much computation at transformation time as possible. Computations fully executed at transformation time are not repeated at runtime. Partial evaluators also perform computations based upon promises about the inputs that a program will receive at run time. In this case, the result is a program specialized for inputs that obey the promises. Partial evaluation has been investigated as an artificial intelligence tool [2, 12], and is a proven technique for creating compilers and compiler generators [11], automatically rederiving important algorithms [9], speeding up computations by two orders of magnitude [4], parallelizing scientific code [3], and optimizing programs [5]. The transformations responsible for these impressive results include removal of the programmer's data and control abstractions, sharing of computations that would otherwise be duplicated at runtime, and pre-execution of computations.

This paper presents the use of graphs as an intermediate representation during partial evaluation. This technique, pioneered by Berlin [3], is responsible for the speedups reported by Weise and Berlin [4]. In that research, graphs could only express straightline programs, and were only motivated by their ability to expose parallelism. This paper motivates graphs from a much larger context, extends them to express arbitrary programs, and carefully describes where their power comes from. Using graphs as the intermediate representation has the following benefits:

- There is no fear of code duplication to inhibit partial evaluation.
- All objects can be constructed at runtime, yielding the important ability to both reduce expressions and leave them residual.
- Type information can be easily propagated.
- Parallelism is exposed because the semantics of graph evaluation is inherently parallel. (This benefit is not discussed in this paper. For more information see [3].)
- Control flow can be expressed as data flow, so that side-effects do not have to be treated specially.
- They are a well understood technology, being used in standard compilers, and as a program representation form in graph reduction machines that implement lazy functional languages.

In our method, partial evaluation performs a nonstandard interpretation of the program. Where standard evaluation maps a program and an environment into an answer, partial evaluation maps a program and an environment into a graph. During partial evaluation special values, which we call *partial values*, represent the actual values that a program will see at run time. Partial values also represent nodes in the graph that partial evaluation produces: creating a partial value adds a node to the graph. The primitive functions, such

Graphs as an Intermediate Representation for Partial Evaluation

Daniel Weise

Technical Report: CSL-TR-90-421
(Also FUSE Memo 90-1)

March 1990

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract: We have developed a fully automatic partial evaluator, called FUSE, for the functional subset of the Scheme programming language. FUSE operates differently from other partial evaluators: it transforms programs into graphs rather than into program text. A separate program transforms the graphs into executable code. This paper shows how using graphs solves difficult problems encountered with conventional partial evaluation technology such as code duplication, type propagation, and premature decision making. The major benefit of graphs is the ability to both reduce an expression at partial evaluation time and to have the expression appear in the transformed program.

Key Words and Phrases: Partial Evaluation, Program Transformation, Compilers, Symbolic Execution

Graphs as an Intermediate Representation for Partial Evaluation

Daniel Weise

Technical Report No. CSL-TR-90-421

March 1990

This research has been supported in part by NSF contract MIP-8902764, and in part by Advanced Research Projects Agency, Department of Defense, Contract N00014-87-K-0828.