Practical Partial Evaluation

by

Rajeev Surati

S.B. Massachusetts Institute of Technology (1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author ....
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by ....
Harold Abelson
Class Of 1922 Professor and Macvicar Teaching Fellow
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Practical Partial Evaluation

by
Rajeev Surati

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degree of
Masters of Science in Electrical Engineering

Abstract

Partial evaluation techniques have been known to yield order of magnitude speedups of real-world applications. Despite this tremendous potential, partial evaluators are rarely put to practical use. This is primarily because it is not easy for users to extend partial evaluators to work on their specific applications. This thesis describes **Blitzkrieg**, a user-extensible online partial evaluator designed to make partial evaluation more accessible to a wider class of users and applications.

Blitzkrieg uses an object system to simplify the implementation of the partial evaluation system and to allow users to easily express their own domain-specific optimizations.

The viability of **Blitzkrieg** is shown by applying it to the Stormer integrator on a 6-body problem (achieving a factor of 610 speedup) and the Runge-Kutta integrator on the same problem (a factor of 365.5 speedup). In addition, the flexibility of the approach is demonstrated by applying **Blitzkrieg** to handle port input programs, achieving a factor of 1.35 speedup on a representative program. Finally, **Blitzkrieg**'s ability to do user-expressed domain specific optimizations is demonstrated on a graphics application, achieving a factor of 4.04 speedup.

Thesis Supervisor: Harold Abelson
Title: Class Of 1922 Professor and Macvicar Teaching Fellow
Department of Electrical Engineering and Computer Science

Practical Partial Evaluation

by
Rajeev Surati

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degree of
Masters of Science in Electrical Engineering

Abstract

Partial evaluation techniques have been known to yield order of magnitude speedups of real-world applications. Despite this tremendous potential, partial evaluators are rarely put to practical use. This is primarily because it is not easy for users to extend partial evaluators to work on their specific applications. This thesis describes **Blitzkrieg**, a user-extensible online partial evaluator designed to make partial evaluation more accessible to a wider class of users and applications.

Blitzkrieg uses an object system to simplify the implementation of the partial evaluation system and to allow users to easily express their own domain-specific optimizations.

The viability of **Blitzkrieg** is shown by applying it to the Stormer integrator on a 6-body problem (achieving a factor of 610 speedup) and the Runge-Kutta integrator on the same problem (a factor of 365.5 speedup). In addition, the flexibility of the approach is demonstrated by applying **Blitzkrieg** to handle port input programs, achieving a factor of 1.35 speedup on a representative program. Finally, **Blitzkrieg**'s ability to do user-expressed domain specific optimizations is demonstrated on a graphics application, achieving a factor of 4.04 speedup.

Thesis Supervisor: Harold Abelson
Title: Class Of 1922 Professor and Macvicar Teaching Fellow
Department of Electrical Engineering and Computer Science

Acknowledgments

I would like to acknowledge the help, advice and encouragement I received from Elmer Hung, Natalya Cohen, and Dr. Kleanthes Koniaris, Dr. Andrew Berlin, Dr. Erik Ruf, Prof. Olivier Danvy, Nate Osgood, Chris Hanson, Dr. Stephen Adams, Dr. Guillermo Rozas, Jason Wilson, Luis Rodriguez, Daniel Coore, Rebecca Bisbee, Thanos Siapas, Michael Blair, and Prof. Mitchell Trott, Prof. Hal Abelson, and Prof. Gerry Sussman,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Need for Blitzkrieg

Partial evaluation is a program optimization technique that attempts to reduce a program to the "essential" computations necessary to generate the desired output. In particular, given information about a program's inputs, partial evaluation (PE) can be used to specialize a program so that it performs optimally for a specific type of input data, as depicted in Fig. 1-1.

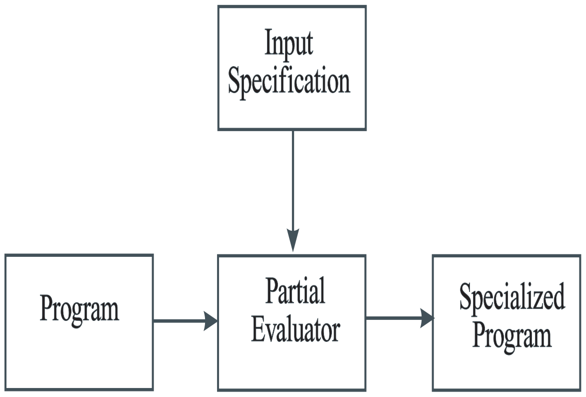

Figure 1-1: The Partial Evaluation Process.

As a simple example, let us consider a generic `dot-product` function that might come from a scientific library:

```
(define (dot-product x y)
  (if (and (vector? x)
           (vector? y))
      (let ((xvlength (vector-length x))
            (yvlength (vector-length y)))
        (if (= xvlength  yvlength)
            (let loop ((i 0)
                       (sum 0))
              (if (= i xvlength)
                  sum
```

```
                (loop (+ 1 i)
                      (+ (* (vector-ref x i)
                            (vector-ref y i))
                         sum))))
          (error "vector-lengths differ")))
    (error  "dot-product:input type incorrect"))).
```

While the function is very general and safe, let us assume that the user knows that `x` and `y` will always be vectors of length two. In this case, the partial evaluator (PE) might produce a function that looks as follows, which is also what an experienced programmer might produce, to be automatically be used instead of `dot-product`:

```
(define (new-dot-product x y)
  (+ (* (vector-ref x 0) (vector-ref y 0))
     (* (vector-ref x 1) (vector-ref y 1))))
```

The specialized procedure `new-dot-product` does not type check the input vectors, take the lengths of the input vectors, compare these lengths, do computations on loop variables, nor add `0` to the result; these operations are only done in `dot-product`. `dot-product` has been unrolled, and many of the the subcomputations formerly done at run time in have been done at compile time for `new-dot-product`, because the user has decreed that the inputs to the specialization will *always* be vectors of length two. With such an assurance from the user, a substantial amount of work that would have been done at run time is done at compile time, thereby reducing the total execution time. Even further optimization is possible: if it is known that the vector is composed of floating-point vectors, then generic multiplications (`*`) can be replaced with floating-point specific operations (`flo:*`), etc.

It has already been shown that partial evaluation techniques are capable of producing orders of magnitude speed-ups on real-world applications ([6], [20]). Despite this tremendous potential, however, partial evaluators are rarely used to support real-world applications, and this is due to four major reasons.

First of all, traditional partial evaluators are not easily extended to work on most user's applications. Extensibility is important because the designer of a partial evaluator can not be expected to write a partial evaluator that handles and optimizes everything a user desires. To make a real difference in system performance one must often teach the PE about specific optimizations for their applications. If one introduces a new data-type, there should be a way to teach the PE about it. For example, let us suppose that we have added complex numbers to a system; the PE must be taught to realize that complex zero plus a real x is qual to x, etc. Opportunities for such optimizations are often made visible by partial evaluation, and ease of extensibility is a critical consideration for a user.

A second problem is that partial evaluators often have difficulty with imperative languages, and they can get confused and produce incorrect programs—a wrong answer quickly delivered is not particularly desirable. However, it is often possible to detect these cases and issue a warning.

A third problem is that partial evaluators are often difficult to control. As they examine the source program they can get confused and fail to terminate. The user must often guide them with expert knowledge about the PE's design, and few people are this tolerant.

A fourth problem is that liberal partial evaluation often results in extremely large binaries that are not warranted by their increase in performance.

However, of all of the problems that I list, it strikes me that the first one—lack of extensibility—is the most significant, and therefore the most important to address.

I present **Blitzkrieg**, a user-extensible partial evaluator that works on a large class of programs. Three properties of **Blitzkrieg** make it easily extensible. First, it is easy to express new input specifications previously unknown to the system. In other words, one can create their own datatypes and expect to extend **Blitzkrieg** to be able to partially evaluate programs with varying levels of unknown information with respect to that type of input. Second, the user can easily make his procedures behave differently for combinations of known inputs and unknown values. Finally, in addition to employing standard partial evaluation techniques, **Blitzkrieg** can take advantage of application-specific optimizations based on information supplied by the user. That is, information derived from inputs that are unknown at compile time is represented by placeholders which contain a documented representation of information necessary to generate their actual value at run time. This representation may be traversed and analyzed to perform these domain-specific optimizations.

To show the viability of **Blitzkrieg**, I compare the execution times of a set of partially evaluated programs against their original versions. I applied **Blitzkrieg** to the 6-body problem—a Stormer integrator (610$\times$ speedup) and a Runge-Kutta integrator (366$\times$ speedup). In addition, I demonstrate the flexibility of the approach by extending the system to handle fixed-format port input programs, achieving 1.35$\times$ speedup on a representative program. Finally, **Blitzkrieg** achieves 4.04$\times$ speedup on a graphics application by performing application-specific optimizations.

1.2 Description

Blitzkrieg is an online partial evaluator—it decides whether to residualize (save for run time) or continue to specialize a computation during the specialization phase. It performs source-to-source transformations, meaning that its input is a Scheme program, and its output is another often larger but faster Scheme program.

Before using a program, the user partially evaluates it with **Blitzkrieg**, taking care to provide as much information as he can about the program's inputs. **Blitzkrieg** partially evaluates an input program by passing it to the MIT Scheme interpreter that I extended to perform abstract interpretation.

Throughout abstract interpretation, unknown inputs are represented by *placeholder objects* [6]. The presence of a placeholder in a computation indicates that the computation cannot be performed right away; therefore, the system residualizes the computation until run time. Initially, only unknown inputs to the entire program are represented by placeholders. As abstract interpretation is performed recursively on subcomputations of the initial program, and those subcomputations that contain placeholders among their inputs are themselves residualized—that is, a new placeholder is made for any subcomputation with placeholder inputs.

Placeholder objects inherit properties from a special placeholder class. They also take on any other characteristics of classes that the user specifies that his new placeholder class inherits from. Each placeholder contains information about how it was produced (i.e., its internal structure). For example, a placeholder that represents an input to the program is simply indicated as an input placeholder, and a placeholder obtained by residualizing an expression that multiplies two other placeholders has this information associated with it. Thus, placeholders provide a mechanism for residualizing computation while preserving the program's structure. This structural information stored within placeholders can be traversed and used during partial evaluation.

A user need do only two things to extend the partial evaluator to handle his application. First, he needs to construct placeholder subclasses for his application's data structures. This can be done by simply creating a new class (a mixin [18]) that inherits from both his data structure's class and the placeholder class. Second, he needs to identify the procedures that do actual work within his application that will be given placeholder inputs, and redefine these procedures to residualize on the proper placeholder inputs. When residualizing the code, the user can take advantage of the program's structural information available in the placeholders to perform domain-specific optimizations. This is demonstrated in the example below.

1.3 Motivating Example

In order to demonstrate the need for a system such as **Blitzkrieg**, consider the example of computing two-dimensional graphical transformations. Two-dimensional graphical transformations such as rotate, scale, and translate can be performed on a point (x, y) via matrix multiplication. Specifically, a two-dimensional point (x, y) is represented as a vector $\begin{bmatrix} x & y & 1 \end{bmatrix}'$, and is operated on by a transformation matrix T. For a rotation angle θ, scaling factor m, and translation by (x_t, y_t), the transformation matrix is

$$T = m \begin{bmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{bmatrix}.$$

Thus, in this representation our two-dimensional graphical transformation is simply a linear transformation in 3 space.

We can define a procedure to perform this transformation on a point as follows:

```
(define (2D-transformation scale rotation translate-x translate-y point)
    (if (point? point)
        (apply-2D-transform
         (make-transformation-matrix
          scale
          rotation
          translate-x
          translate-y)
         point)
        (error "Point argument not point")))
```

apply-2D-transformation takes two arguments, a matrix and a point and returns a new point that is the result of applying the aforementioned linear transformation input as the matrix to the input point. It essentially is a procedural abstraction for matrix-vector multiply.

It is often convenient to use procedural composition in order to implement successive transformations applied to a point. For example, if T_1 and T_2 are two transformation matrices, we might want to transform the point x by T_2, and then transform the result by T_1, effectively computing $T_1(T_2x)$. This means that two applications of apply-2D-transform (and thus two matrix multiplies) would be required to produce the result. However, note that $T_1(T_2x) = (T_1T_2)x$. If T_1 and T_2 are known at compile time, the matrix multiplication T_1T_2 can be done at compile time, and only a single matrix multiplication is required at run time. (Note that an experienced programmer might define $T_3 = T_1T_2$, and then use T_3x all over the program, but this would defeat the purpose of the library functions; in some sense, we wish for the PE to help us in the conflict between abstraction and performance.)

In a standard partial evaluation system it would be very difficult for a user to specify the above optimization. In Blitzkrieg, however, a user can write procedures to inspect the placeholders for information on how the placeholders were generated. This information can be traversed and used to do the domain-specific optimization. The data abstraction for the generating information is well-defined, hence it easy for a user to write his own procedures to traverse these data structures as well as determine various properties of them. In the following example we add a method to apply-2D-transformation for point placeholders to capitalize on the possibility that apply-2D-transform may be effectively composed and thus actually perform the matrix-multiply at compile time.

Assume that the user wrote generated-by-apply-2D-transform? using the system's data abstractions to check the placeholder's generator information in order to determine whether the placeholder was generated by a call to apply-2D-transform. This is possible because the placeholder contains a representation of the residualized code indicating how the placeholder was generated. The systems data abstraction will be explained later— the details are irrelevant to explain this concept. What is important, however, is with such an abstraction there is a means by which any user can capitalize on the generator information. Also, assume that the user has written generator-matrix, using the system's data abstractions, to retrieve from the apply-2D-transform-generated input point placeholder, that last matrix argument. Similarly, assume that the user has defined generator-point, using the system's data abstractions to retrieve from `apply-2D-transform`-generated input point placeholder the initial point placeholder. In **Blitzkrieg**, a user can simply express the optimization:

```
(define-method apply-2D-transform ((matrix <matrix>)
                                   (point  <point-placeholder>))
  (make-new-placeholder <point-placeholder>
                        (if (generated-by-apply-2D-transform? point)
                            (residualize apply-2D-transform
                                         (matrix-multiply  matrix
                                                           (generator-matrix point))
                                         (generator-point point))
                            (residualize apply-2D-transform
                                         matrix
                                         point))))
```

As mentioned above, the unknown values in the system are placeholders. The above method is used if the two inputs to `apply-2D-transform` belong to the `<matrix>` and `<point-placeholder>` classes, respectively. Because `matrix` belongs to the `<matrix>` class, its elements are all explicitly known, though some may be placeholders. The procedure returns a new point placeholder, and it checks whether the point placeholder was generated by a call to `apply-2D-transform`. If so, then the current transformation matrix and the transformation matrix associated with the point placeholder (the `generator-matrix` of the input point placeholder) are multiplied to generate a new transformation matrix. The `apply-2D-transform` procedure is then residualized with the new transformation matrix and the original point from the point placeholder. On the other hand, if the input point placeholder was not generated by a call to `apply-2D-transform`, then `apply-2D-transform` simply residualizes the call to `apply-2D-transform` on the current input matrix and input point placeholder. This simple extension of the partial evaluator is all that is needed to make successive 2D transforms efficient in a graphics system.

Interestingly, performance can also be improved in terms of reducing run-time heap allocation. Note that originally the implementation does a lot of memory allocation—one matrix per each `apply-2D-transform`. With this optimization, what was a linear amount of `cons`ing with respect to the number of invocations of `apply-2D-transform`s was turned into a constant-time operation. So, this optimization has not only reduced the multiplications and additions necessary, it greatly reduces memory usage. There are secondary effects to be derived in terms of performance that come from reducing memory usage per invocation. Reducing memory usage in this manner will reduce the amount of garbage collection necessary thereby improving performance as measured by the wall-clock.

The above example shows the power of having a partial evaluator that is easily extensible and allows its users to express their own application-specific optimizations. The abstractions used in the original code written by the user (in this case, composable 2D-transformations) make the program easy to write and easy to understand; the partial evaluator is then employed to ensure that the use of abstraction does not result in sacrificing efficiency of the program. Normal compiler analysis would not even make the possibility of this optimization visible.

In **Blitzkrieg**, partial evaluation removes the obstacle of abstraction, making it possible for the domain-specific optimizations to be exploited. It is then up to the user to express such optimizations so that they may be performed. **Blitzkrieg** provides a simple means for the user to extend the partial evaluation to his application's input data and to express further application-specific optimizations. In the example above, there is no way a standard partial evaluator could take advantage of the matrix multiply associativity optimization with the point-placeholder method added for `apply-2D-transform` ; the user has to specify that such an optimization is permissible.

Granted, the original code `2D-transformation` can be written in a more clever way to take advantage of matrix associativity. The point of this example, however, is that rewriting the program in clever and non-obvious ways is not necessary in order to express application-specific knowledge. Furthermore, the approach demonstrated in the example is not only easier to implement, but sometimes it is the only viable solution. For example, people often compose procedures from a library provided in the system. Usually, several such procedures composed together perform a lot of redundant and unnecessary computation like type checking and the unpacking and packing of data-abstractions for the sake of nice data abstraction for the user. In between all of this the essential calculations are done!

The user does not want to spend his time changing the library procedures to get rid of these redundancies. Not only would he have to get rid of the redundancies, he would actually have to understand the low-level details of the library to figure out what these redundant operations are; on the other hand, he (or the library's implementor) can easily extend **Blitzkrieg** to express several domain-specific optimizations, and then partially evaluate his code. Much like the above example where the optimization for `apply-2D-transformation` is done with a lower-level `matrix-multiply`, the implementor could encode the composition optimization in terms of the lower level procedures that occur between the type checking, unpacking and packing of objects of the library abstractions available to the user. Thus the composition would eliminate all the extraneous type checks, as well as the unpacking and packing of objects and be expressed in terms of the lower-level procedures that do the actual data manipulation. The compositions will then be as good as if the library implementor had written them—automatically eliminating redundant and useless computation.

1.4 Overview

In this chapter I have explained the motivation for writing **Blitzkrieg**, and demonstrated the system's operation and capabilities on a simple example. Chapter two describes the design of **Blitzkrieg** and its extensibility. Chapter three examines the performance improvements that result by applying **Blitzkrieg** to several real-world problems, including the 6-body solver (a Stormer integrator is 610$\times$ faster, and a Runge Kutta integrator is 366$\times$ faster), a program for fixed-format port input (1.35× faster), and a graphics application (4.04× faster). Chapter four discusses related work, and chapter five gives conclusions.

Chapter 2

Blitzkrieg : Design and Implementation

2.1 Overview of Blitzkrieg

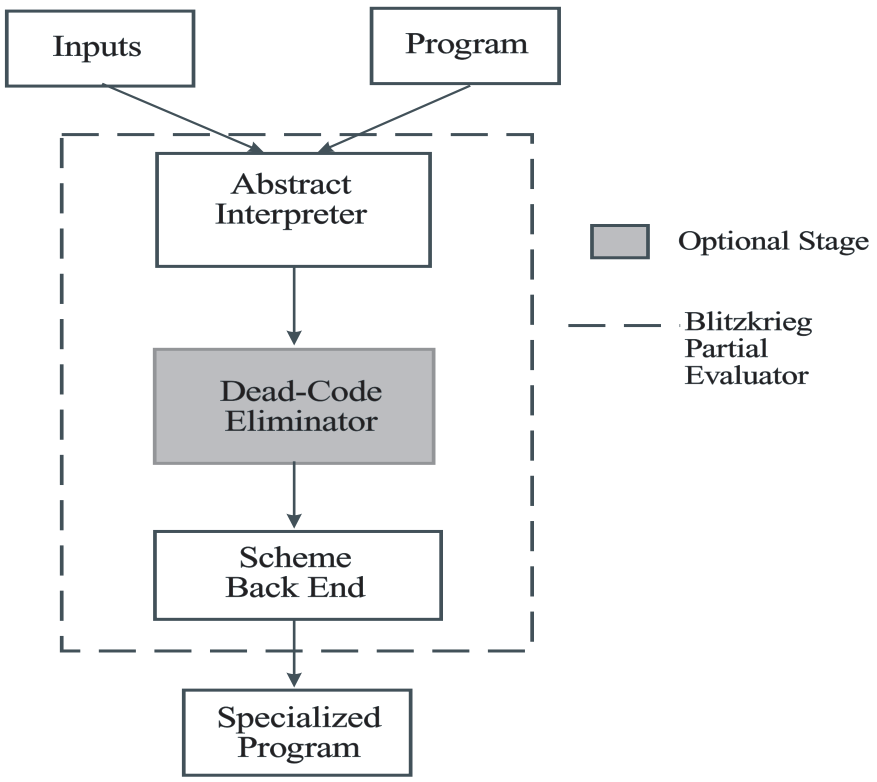

Figure 2-1: Internal structure of the **Blitzkrieg** partial evaluator.

As depicted in Fig. 2-1, the **Blitzkrieg** partial evaluator has three major components. The first is the abstract interpreter. When handed a program and information about the program's inputs, it evaluates as much of the program as possible given the information available, and replaces the rest of the code (that is, the code that represents computations with unknown inputs) by placeholder objects. Thus, the output of the abstract interpreter represents the computations carried out by the program in terms of operations on placeholder objects. The next module, the (optional) dead-code eliminator, deletes any code produced by the abstract interpreter that does not contribute to the computation of the relevant outputs of the program. This module exists only to avoid unnecessarily large programs. Finally, the Scheme back end translates the output of the dead-code eliminator into a valid Scheme program. Thus, **Blitzkrieg** performs a source-to-source transformation, taking as input a Scheme program and information about said program's inputs, and outputting a partially evaluated Scheme program that efficiently computes the same answer as the original program when evaluated with inputs that meet the given input specifications.

As an illustrative example,[1] suppose that one would like to use **Blitzkrieg** to partially evaluate

```
(increment-then-double x)
```

where `x` is a floating-point number, and `increment-then-double` is given by the following Scheme code:

```
(define (increment-then-double x)
  (* 2 (1+ x)))
```

In order to accomplish this, the user must first encode information about the type of input by creating a floating-point placeholder:

```
(define input (make-new-placeholder <flonum-placeholder> 'x))
```

Partial evaluation of the desired expression can then be accomplished as follows:

```
(partial-evaluate (lambda () (increment-then-double input)))
```

As described above, partial evaluation proceeds in three stages. The abstract interpreter of **Blitzkrieg** takes the input specification `input` and the program `increment-then-double`, and produces a data structure, designated here by `output-placeholder`, that represents the computations carried out by
`increment-then-double` in terms of operations on placeholder objects:

```
(show output-placeholder)
Result: (<flonum-placeholder 3>
          (generator
             (generator-combination `(flo:* ,system-global-environment)
                                    2.0
                                    <flonum-placeholder 2>)))
```

[1]The example described here presents a greatly simplified view of **Blitzkrieg** s output. I do this in order to illustrate the main functionality of each of **Blitzkrieg** s modules without cluttering the presentation with machine-readable representations.

The details of `output-placeholder` are not important for now. In section 2.2 I explain the structure of `output-placeholder` and how the abstract interpreter goes about generating it.

In addition to the `output-placeholder` data structure, the abstract interpreter generates an execution trace for `increment-then-double`. As implied by its name, the execution trace keeps track of the order in which the placeholders present in the program get executed at run time (once the inputs become known). The details of the particular execution trace for our example are also explained in Sec. 2.2.

The results of the abstract interpretation (both `output-placeholder` and the execution trace) are sent to the dead-code eliminator. The dead-code eliminator eliminates those instructions in the execution trace that do not contribute to the computation of the `output-placeholder`.

Finally, the Scheme back end generates optimized Scheme code for `increment-then-double`, given the revised execution trace produced by the dead-code eliminator, `input`, and `output-placeholder`. The resultant code implements the instructions contained in the execution trace, returning the value represented by the `output-placeholder`.

Now that I have given a high-level idea of how the system operates, I shall now explain the use, design, and implementation of the various modules in greater detail. In the following sections I discuss the three stages of **Blitzkrieg** depicted in fig. 2-1: the abstract interpreter, the dead-code eliminator, and the Scheme back end.

2.2 The Abstract Interpreter

2.2.1 Overview of the Abstract Interpreter

Blitzkrieg uses the underlying MIT Scheme interpreter (that has been extended for this purpose) along with the MIT Scheme Object System (SOS) to implement the abstract interpreter. As explained in the previous section, the abstract interpreter outputs two objects: the data structure representing computations carried out by the program in terms of the placeholder objects (such as `output-placeholder` in the previous example), and the program's execution trace.

Closer examination of the placeholders in the `increment-then-double` example will yield greater insight into what actually happens during abstract interpretation:

```
(show input)
Result: (<flonum-placeholder 1> (generator x))

(show output-placeholder)
Result: (<flonum-placeholder 3>
         (generator
            (generator-combination `(flo:* ,system-global-environment)
                                    2.0
                                    <flonum-placeholder 2>)))

(show <flonum-placeholder 2>)
Result: (<flonum-placeholder 2>
(generator
   (generator-combination `(flo:1+ ,system-global-environment)
                          <flonum-placeholder 1>)))
```

The generator of a placeholder contains information about the origin and history of the placeholder. Thus, all the relevant residualized program information for each of the above placeholders is stored in its generator. The details of generator representation are discussed later in this section.

Note that in the above example, the `output-placeholder` object is the result of a floating-point multiply between the constant 2.0 and `<flonum-placeholder 2>`. Comparing the original code of `increment-then-double` to `output-placeholder`, it is clear that the abstract interpreter was able to use information about the input type to turn an application of `*` into the more specialized call to `flo:*`, while also coercing the integer 2 to the floating-point number 2.0.

As mentioned earlier, the execution trace generated by the abstract interpreter is an ordered list of placeholders; it reflects the order in which the computations represented by the placeholders should be performed at run time. Every time a new placeholder is created, it is appended to the execution trace. For example, the execution trace for `increment-then-double` is a list of `<flonum-placeholder 1>`, `<flonum-placeholder 2>`, and `<flonum-placeholder 3>` in that order, reflecting the fact that `x` has to be evaluated before `(1+ x)`, which in turn has to be evaluated before `(* 2 (1+ x))`.

The execution trace becomes somewhat more complicated to deal with if there are data dependencies. In this case, however, a similar result can still be achieved using labels like `if`'s and `goto`'s, making certain to annotate this additional information while abstractly interpreting code containing data-dependent conditionals. Data dependency is discussed in more detail later in this section.

With this overview of the abstract interpretation process in mind, we now describe in greater detail how the abstract interpreter was designed and implemented.

2.2.2 Design and Implementation of Abstract Interpreter

In this section, I investigate how the design of the abstract interpreter and using an object system makes **Blitzkrieg** extensible. I discuss the object system, the implementation of placeholder classes, the process of extending the system, and the mechanism used by the system to handle data dependencies.

The Object System

Three properties of the **Blitzkrieg** system make it easily extensible. First, it is easy to create placeholders that represent types previously unknown to the system. Thus, the user can create new data structures and placeholders for those data structures, and use the placeholder to partially evaluate his program. Second, the user can easily make his procedures behave differently for known inputs and placeholder inputs. Third, there is a well-defined interface for dealing with the placeholder's representation of how it was generated.

These three features are implemented through the use of an object system. Class hierarchies are used to obtain an extensible typing system. Every object in the system belongs to (or "is an instance of") a certain class; for example, there is a `<flonum>` class for floating-point numbers, and a `<vector>` class for vector objects. A special `<placeholder>` class is defined for placeholder objects. Classes for placeholders of specific types are obtained by combining the `<placeholder>` class with other classes (as per a mixin in CLOS); for example, the `<flonum-placeholder>` class inherits from both the `<placeholder>` class and the `<flonum>` class. Likewise, new types and the placeholders for those types can be built on top of existing classes.

The difference in behavior of procedures for known versus placeholder inputs is implemented via generic functions, also available in the SOS object system. Once a procedure is declared to be generic, different methods can be written to deal with different classes of inputs. Thus, for instance, two methods for the procedure `int:1+` can be implemented, one for an integer input, another for an integer placeholder input.

The actual object system used in **Blitzkrieg** is the MIT Scheme Object System (SOS), described in [16]. SOS was derived from Kiczales's Tiny CLOS ([19]) that was loosely derived from CLOS [24][18]. SOS contains both a class inheritance mechanism and generic functions, allowing us to achieve the desired functionality described above. In addition, SOS has the nice property that its syntax is rather well known because it is a subset of the one for CLOS. Thus, it is easy for the users to learn how to extend **Blitzkrieg** to their application domains.

The Placeholder Class

As evident from the overview, the `<placeholder>` class plays an important role in the abstract interpretation phase of the partial evaluator. I now examine the properties of this class in more detail.

The `<placeholder>` class is defined as follows:

```
(define-class <placeholder> (<object>) generator)
```

That is, it inherits from the system's base level `<object>` class and has one slot named `generator`. This slot contains information about how the placeholder was generated. The slot accessor, `placeholder-generator`, is used to obtain the contents of the generator slot.

New placeholders are created with a construct `make-new-placeholder` that takes two arguments: the class of the placeholder to make, and the generator to use.[2] Two kinds of generators can be specified by the user: a Scheme symbol, and a generator combination. I discuss each possibility below and its meaning in terms of residualized code.

[2]It is worth mentioning that, besides the two required arguments, `make-new-placeholder` accepts any number of optional arguments. These optional number of arguments are used to initialize any additional slots the placeholder class in question might have. Thus, it is easy to define classes inheriting from the `<placeholder>` class that have additional slots, and to create instances of these classes.

A placeholder with a Scheme symbol in its generator slot represents an `input` to a program. It is created by the user, and used during partial evaluation in place of the actual program input specified at run time. An example of this kind of placeholder is `input`, defined in sec. 2.1. From its printed representation in this section's overview, it is clear that `input` indeed has a symbol, `x`, in its `generator` slot.

A placeholder with a generator combination in its `generator` slot represents a residualized procedure call. It is created when one or more of the arguments to the procedure is a placeholder, and thus the computation has to be saved for run time. The generator combination within the newly created placeholder contains information about the procedure and its arguments. More specifically, a generator combination is a tagged list; besides the `generator-combination` tag, it contains the name of the procedure to be residualized (together with the environment in which the procedure is accessible), and the arguments to the procedure. The arguments can be any Scheme objects or placeholders. For instance, the generator slot of `<flonum-placeholder 2>` in this section's overview contains a generator combination; in this case, the residualized call is to procedure `flo:1+` defined in the `system-global-environment`, with one placeholder argument, `<flonum-placeholder 1>`. While this particular generator combination is system-generated, I shall show examples of user-generated combinations later in this section.

The procedure and arguments within a generator combination can be accessed with `generator-combination-procedure` and `generator-combination-arguments`, respectively. In addition, a generator combination can be identified by the predicate `generator-combination?`. These accessors and predicates can be used to traverse the generator slot of a placeholder for performing domain-specific optimizations.

Besides the generators that can be created and manipulated by the users, there are two kinds of generators that are system-generated. The first one of these, a conditional generator, represents the result of having a data-dependent conditional; it indicates that a conditional must be resolved at run time. The second one, an alias generator, is a result of performing a placeholder class conversion. The need for both of these kinds of generators is explained in greater detail later in the section, when I talk about data dependencies.

Armed with the object system and the placeholder and generator constructs, I can begin extending the system to perform abstract interpretation in presence of various kinds of placeholders. I shall now discuss how such extensions are implemented.

Extending the System to Integers

To demonstrate how the user can extend **Blitzkrieg**, I show the process by which **Blitzkrieg** was extended to deal with integers. Note, however, that **Blitzkrieg** has already been extended to deal with most of the MIT generic arithmetic system. Thus, the user would not have to implement such an extension himself.

To begin the process, one makes the following incantation:

```
(define-class <integer-placeholder> (<integer> <placeholder>))
```

This creates a class called `<integer-placeholder>` that inherits from both the `<placeholder>` and the `<integer>` classes.

Once the new class is defined, one need only define the primitive integer operations, such as `int:+`, `int:-`, `int:1+`, and `int:*`. I shall implement `int:+` as a representative example.

First, one declares `int:+` to be a generic procedure:

```
(define-generic-procedure int:+ (x y))
```

In the case when both inputs are known, `int:+` should simply add the two numbers. I define a method that handles this case as follows:

```
(define-method int:+ ((x <integer>)
                      (y <integer>))
  ((make-primitive-procedure 'int:+) x y))
```

In the case when one of the arguments is a known integer and the other is an unknown integer (i.e., an integer placeholder), I make a new integer placeholder with a generator combination in the generator slot. The generator combination serves to residualize the call to `int:+`. Note the optimization employed that checks for a zero input.[3]

```
(define-method int:+ ((x <integer-placeholder>)
                      (y <integer>))
  (if (int:zero? y)
      x
      (make-new-placeholder
               <integer-placeholder>
               (make-generator-combination  `(int:+ ,system-global-environment)
                                             x
                                             y))))

(define-method int:+ ((x <integer>)
                      (y <integer-placeholder>))
  (int:+ y x))
```

Finally, when both inputs to `int:+` are placeholders, I always residualize the procedure call until run time:

```
(define-method int:+ ((x <integer-placeholder>)
                      (y <integer-placeholder>))
  (make-new-placeholder
        <integer-placeholder>
        (make-generator-combination `(int:+ ,system-global-environment)
                                    x
                                    y)))
```

[3]In the next chapter, I will show how one can perform further optimizations by traversing the program's graph in the placeholder's generator slot.

As one can see, it's trivial to extend the system to more procedures and objects using the above paradigm. However, there are cases when extending a particular generic procedure to handle placeholders will not work: it is possible to inadvertently express an infinite recursion by overloading a procedure that happens to be invoked by the Scheme system itself. For example, if one is overloading `fix:+` to extend the system, and if the code generated for the generic function `fix:+` happens to also call `fix:+`, there might be a problem. If such a problem occurs, one must carefully define the procedure to do the dispatching to various methods by hand to be certain no infinite recursion is expressed. One can write the PE in such a way as to avoid this problem through proper use of environments.

Data Dependencies

In our discussion has been restricted so far to dealing with programs in which the computation is data-independent. In most real programs, however, abstract interpretation is complicated somewhat by the presence of data dependencies. In particular, if our program contains a conditional with a placeholder predicate, we cannot decide at compile time which branch of the conditional will be taken! Thus, our abstract interpreter needs a mechanism for dealing with data-dependent computations. Essentially, what is meant by "data-dependent computation" is code that has control flow rather than one gigantic basic block of straight-line code.

The mechanism to handle conditionals employed by **Blitzkrieg** is similar to the one described by Berlin in [6]. Berlin's system has several limitations, and some of them are severe. One limitation is that one must be able to execute the consequent of a conditional without affecting the result of executing the alternative, and vice versa, in order for his partial evaluator to work correctly. For example, if an `if` expression sets a global variable to 5 in the consequent and to 7 in the alternative, Berlin's system does not guarantee correctness. **Blitzkrieg** currently has the same requirement.[4] Berlin also requires that the structure of the object in either branch of the conditional be the same. The requirement in **Blitzkrieg** is not quite as strong, as explained below.

The construct used to aid in partial evaluation of data-dependent computations is called `pe-if`; it is a special form used in place of `if`.[5]After evaluating the predicate, `pe-if` determines whether or not the result is a placeholder. If it is not a placeholder, then `pe-if` evaluates the consequent or the alternative just as `if` does. However, if

[4]I believe that with further work this restriction can be lifted. In particular, if all the side-effects are logged (and thus are reversible), it should be possible to set the effected variables, merging the values that are valid during the execution of the consequent and the alternative. Being able to reverse the side effects would go a long way toward extending the class of procedures **Blitzkrieg** can partially evaluate. The name **Blitzkrieg** was derived from this restriction to indicate that the systematic basic approach on most programs is to unroll the entire computation.

[5]Work was done to make it unnecessary for the user to explicitly identify instances where `pe-if` must be used. One could simply replace all occurrences of `if` with `pe-if` in all relevant code, but that would cause a substantial amount of code bloat as the `pe-if` macro expands to a relatively large amount of code. Explicit replacement was therefore the chosen alternative.

the result of evaluating the predicate is a placeholder, then both the consequent and alternative are evaluated. The results of the two evaluations are merged together by a generic procedure, `join-objects`. If the two objects being joined are actually the same (i.e. `eqv?`) object, `join-objects` returns it—the result of the `pe-if` expression is then the object returned by `join-objects`. Otherwise, a new placeholder is formed, and its class is the least upper bound of the classes of the two objects, and its generator slot is empty. The job of creating the new placeholder is finished by `pe-if`, that puts in its generator slot a conditional generator containing the placeholder for the predicate, as well as the results of evaluating the consequent and the alternative. In will be shown in the next section 2.4 how the Scheme backend deals with placeholders that have conditional generators.

Note that since `join-objects` is a generic procedure, the user has the power to override its default behavior by defining his own method for it. This adds to the extensibility of the system, since it means that the user has control over the mechanism for resolving similarities and differences between objects and for merging them.

We demonstrate the use of `pe-if` by implementing `flo:sqrt`, which represents a data-dependent computation. We assume that `flo:sqrt` has been defined to be a generic procedure, and that the method for a known floating-point number has already been implemented. Thus, I only show how `flo:sqrt` works only when given a floating-point placeholder input:

```
(define-method flo:sqrt ((x <flonum-placeholder>))
  (pe-if (flo:negative? x)
         (make-new-placeholder <inexact-complex-placeholder>
                               (make-generator-combination
                                       `((make-recnum '(runtime number))
                                       0.0
                                       `((flo:sqrt system-global-environment)
                                       (* -1. ,x)))))

         (make-new-placeholder <positive-flonum-placeholder>
                               (make-generator-combination
                                       `(flo:sqrt system-global-environment)
                                       x))))
```

Given this definition, suppose one abstractly interpret the expression

```
(flo:sqrt x)
```

where `x` is a floating-point placeholder. The above method for `flo:sqrt` applies, and since evaluating the predicate yields another placeholder, the consequent and the alternative of the `pe-if` expression are evaluated and joined. The result of the abstract interpretation is thus a placeholder of class `<inexact-complex-placeholder>`, the least upper bound of the types of placeholders found in the consequent and the alternative (namely, `<inexact-complex-placeholder>` and `<positive-flonum-placeholder>`). The generator slot of the resultant placeholder contains a conditional generator, which consists of the boolean placeholder for the predicate, the inexact complex placeholder for the consequent, and the positive floating-point placeholder for the alternative.

In the above example, the object returned is not a floating-point placeholder if the sign of the input is not known. This can be troublesome for the user who is trying to speed up his code, for if he knows that the input to `flo:sqrt` is always greater than zero, but doesn't want to take the time to prove this to the system, then he would like to tell the system this information. For the convenience of such a user, a placeholder type coercion procedure, `coerce-placeholder-class`, is provided in the system. It takes two arguments: the placeholder to be coerced, and the class to coerce it to. A new placeholder of the requested class is returned, with the original placeholder (the alias generator mentioned earlier) in the `generator` slot. The system does not try to verify the correctness of this coercion at run time; rather, the user is trusted to be correct. For example, if `x`, and `y` are known to be floating-point numbers in the following fragment of code:

```
(sqrt (+ (* x x) (* y y)))
```

it might be rewritten to remove the data dependency in `sqrt`:

```
(sqrt (coerce-placeholder-class <positive-flonum-placeholder> (+ (* x x) (* y y))))
```

As one can see, data dependency imposes some severe limitations on the applicability of the **Blitzkrieg** system. Nonetheless, a large class of applications can still be partially evaluated within the limits of these restrictions. In the next chapter, several real-world examples of such applications will be shown.

2.3 Dead-Code Elimination

This optional stage removes useless residualized code to reduce the size of the resulting program. Useless code is any code that does not contribute to the generation of the actual output of the partially evaluated procedure. The dead-code eliminator, when given the results after abstract interpretation, finds all the output placeholders and recursively traverses the computation graphs in each placeholder's generator slot marking all the placeholders that are visited. Placeholders not visited are not marked. These unmarked placeholders are deemed useless for computing the output and are eliminated from execution trace of ordered placeholders. As noted above, the Scheme back end uses this list to compute the resultant code. The absence of these placeholders from that list means that no code will be generated for them as desired. This operation is very closely related to garbage collection— objects (placeholders) that aren't deemed necessary are thrown away.

2.4 Scheme Back End

The Scheme back end generates Scheme code based on the `current-placeholder-list` linear execution trace of the program, the input data structures, and the output data structure. Code generated by the back end for the following procedure partially evaluated with respect to a two element list of an integer followed by a floating point number will be used to explain the capabilities and implementation of the back end:

```
(define (sum-list x)
   (reduce + 0. x))
```

Here is the specialized procedure that is output by **Blitzkrieg** :

```
(Define specialized-sum-list
  (letrec ((normal-vector (make-vector 1))
           (flonum-vector (fvc 3))
           (input-placeholder->vector
            (lambda (input normal-vector flonum-vector)
              (let ((input (car input)))
                (vector-set! normal-vector 0 input)
                (let ((input (cdr input)))
                  (let ((input (car input)))
                    (floating-vector-set! flonum-vector
                                          0
                                          input))))))
           (output-vector->output
            (lambda (normal-vector flonum-vector)
              (floating-vector-ref flonum-vector
                                   2)))
           (inner-body
            (lambda(normal-vector flonum-vector)
              (floating-vector-set! flonum-vector
                                    1
                                    (integer->flonum (vector-ref normal-vector
                                                                 0)
                                                     2))
              (floating-vector-set! flonum-vector
                                    2
                                     (flo:+ (floating-vector-ref flonum-vector
                                                                 1)
                                            (floating-vector-ref flonum-vector
                                                                 0))))))
    (lambda(input)
      (input-placeholder->vector input normal-vector flonum-vector)
      (inner-body normal-vector flonum-vector)
      (output-vector->output normal-vector flonum-vector))))
```

`specialized-sum-list`, like `sum-list`, takes one argument as an input. `normal-vector` and `flonum-vector` are vectors that have values associated with the compile time input-placholders copied into them. All subsequent computations are stored in these vectors. `flonum-vector` is used exclusively for floating-point values. `normal-vector`, on the other hand, is used to store everything else.[6]

[6]This differentiation between floating-point and everything else is necessary to remove one level

The procedure `input-placeholder->vector` is generated based on the input at compile time. This procedure at run time traverses the run-time input data structure. When it reaches a value that appeared as a placeholder at compile time, it copies the value into either the `normal vector` or the `flonum-vector` as appropriate. For `specialized-sum-list`, as can be seen above, the back end generated the following code for dealing with an input that is a list of two elements: an integer followed by a floating-point number.

```
(lambda (input normal-vector flonum-vector)
   (let ((input (car input)))
     (vector-set! normal-vector 0 input)
     (let ((input (cdr input)))
       (let ((input (car input)))
         (floating-vector-set! flonum-vector 0 input)))))
```

As one can see the procedure puts the values from input list into the approprate vector.

The procedure `inner-body` is generated from the `current-placeholder-list` execution trace. Since the list is ordered for execution and happens to have the basic blocks demarcated along it, the code generator need only traverse down the list and generate the code based on each placeholder it meets. The back end, as can be seen above generated the following code for this example's `inner-body`:

```
(lambda(normal-vector flonum-vector)
   (floating-vector-set! flonum-vector
                                     1
                         (integer->flonum (vector-ref normal-vector 0)
                                           2))
   (floating-vector-set! flonum-vector
                                     2
                         (flo:+ (floating-vector-ref flonum-vector 1)
                                (floating-vector-ref flonum-vector 0))))
```

As one can see, the above procedure does the desired computation storing the results in the vectors.

The procedure `output-vector->output` is generated at compile time based on the output data structure. It is generated to create the appropriate output data structure with the appropriate values that were put into `normal-vector` and `flonum-vector` by the procedure `inner-body`. The code for our example's `output-vector->output` as can be seen above looks like:

```
(lambda (normal-vector flonum-vector)
        (floating-vector-ref flonum-vector  2))
```

of indirection that would be necesary in MIT Scheme if a normal vector were used for all the data. Floating-point vectors allow the removal of this extra level of indirection and thus allows allows floating-point computation to be much more efficient by reducing the amount of floating point `cons`ing on the heap.

As one can see, the above procedure accomplishes the desired task returning the sum over the list stored in `flonum-vector` by referencing it from the `flonum-vector` where it was left after `inner-body` was called.

The previous text describes the basic characteristics of code from the Scheme back end, as well as disussing how it is generated. It is pretty straight-forward process. Note that in addition to these capabilities, the back end can also create a procedure for iteration that in the special case where the input data structures look exactly like the output data structures, circumventing the process of transferring the data from the input into the intermediate vectors and then back out, by directly copying the relevant values from the ends of the vectors back to the front as new inputs.

2.5 Chapter Summary

In this chapter the design and implementation of **Blitzkrieg** were discussed—the abstract interpreter, the dead-code eliminator, and the Scheme back end. In the next chapter the effectiveness of **Blitzkrieg** is discussed and shown on several real-world applications.

Chapter 3

Experimental Results

In this chapter, the results of applying **Blitzkrieg** to several real-world applications are discussed. There are three properties of **Blitzkrieg** that are illustrated in this section: **Blitzkrieg**'s viability as a partial evaluator, **Blitzkrieg**'s extensibility, and **Blitzkrieg**'s ability to allow users to express domain-specific optimizations. Experiments to demonstrate these properties are performed on three classes of programs: four programs that perform numerical integrations of the solar system, a fixed format input port program, and a program that implements graphical 3D-transformations.

3.1 Numerical Integrations of the Solar System

I first present results describing **Blitzkrieg**'s performance on optimizing programs that numerically integrate the solar system. The particular programs involved were used by Gerald Jay Sussman and Jack Wisdom [27] to make a landmark scientific discovery concerning the chaotic nature of solar system dynamics. The numerical code implementing the desired integrations is data independent and consists primarily of floating-point computation instructions. The applicability of partial evaluation to this problem, as well as the problem's scientific importance, are discussed in a recent paper by Surati and Berlin [4]. Experimental results of partially evaluating several solar system integration programs have been previously presented by Berlin [6].

In our experiment, four different programs were partially evaluated: a 6-body 4th order Runge-Kutta Integration, a 9-body 4th order Runge-Kutta Integration, a 6-body 13th order Stormer Integration, and a 9-body 13th order Stormer Integration. These are exactly the same programs as used in [6]. Thus, the experiment provides a point of comparison between **Blitzkrieg** and Berlin's partial evaluator, supplying an excellent test of **Blitzkrieg**'s performance as a partial evaluator.

3.1.1 Performance Measurements

Measurements were made of the actual speedup **Blitzkrieg** was able to achieve on the various integration algorithms. The timings represent 3000 iterations of the inner loop of each integrator. The results are shown in Table 3.1. Two sets of measurements were

Performance Measurements					
Problem Desc.	Compiled CScheme	Specialized Program (with boxing)	Specialized Program (no boxing)	Speedup over Compiled	Speedup Berlin '89
6-body RK	317.5s	2.31s	.52s	610	38
9-body RK	584.8s	*	*	*	39
6-body ST	120.6s	1.27s	.33s	365.5	**
9-body ST	212.8s	*	.56s	380	**

Table 3.1: Performance Measurements. Timings are in seconds (s).

The above measurements were done on an HP9000/735 running Scheme 8.0 and the alpha release of its compiler. The Scheme process required a 54.7MB heap in order to compile the programs because the output procedures are so large that the compiler has difficulty dealing with them! This difficulty can be surmounted, however, by breaking up the procedures into smaller fragments that the compiler can handle. A "*" denotes computations that could not be compiled because of insufficient heap for compilation; "**" denotes information that Berlin was also unable to measure due to similar problems, as explained in [6].

taken. The first set of timings measures the speedup from partially evaluating the code with boxed floating-point (FP) numbers (i.e., heap-allocated FP numbers that are tagged and require one level of indirection to read or write read from memory). These timings are labeled "Specialized Program (with boxing)," and are presented in order to make comparisons with timings made by Berlin [6], who also used boxed FP numbers. The second set of timings, labeled "Specialized Program (no boxing)," takes into account the additional speedup gained from tuning the system for FP numbers; in this case, no boxing and unboxing of FP numbers is done at run time. (This optimization for FP numbers was done in the Scheme back end of **Blitzkrieg**and is not a part of its extensibility; rather, it was done to improve **Blitzkrieg**'s utility.)

One of the comparisons made during this experiment was between the speedups with and without boxing of floating-point (FP) numbers. In [6], Berlin predicts a factor of 4 speedup if memory allocation due to boxing and unboxing of FP numbers is eliminated. FP vectors in MIT Scheme enable one to perform this optimization. From table 3.1 one can see that on average, the speedup achieved by using FP vectors was about a factor of 4.1. [1], which is in line with Berlin's prediction. One might also note that, according to [2] and [21], when partially evaluated code consists of long streams of FP instructions, one can expect at least a 20 to 30 percent speedup if instruction scheduling is performed and values that are used once are not stored or loaded, as is the current default in **Blitzkrieg**. These estimates are further supported in [26], where a factor of 6.2 speedup of the 9-body Stormer computation was achieved after partial evaluation by carefully scheduling the computation onto a VLIW parallel processor in order to take advantage of the fine-grain parallelism available in this

[1]from 3.8 speedup on 6-body Stormer and 4.4 speedup on 6-body Runge-Kutta

Optimization Effects		
Problem Description	Initial Operation Count	Operations after Arith. Optimiz.
6-body RK	2113	2113(−0)
9-body RK	4439	4343(−96)
6-body ST	1227	1208(−19)
9-body ST	2155	2103(−52)

Table 3.2: Optimization Effects on *N-Body* Integrations.

computation.

I now compare the performance of **Blitzkrieg** with Berlin's experimental results. As can be seen in table 3.1, because of memory constraint problems associated with the Scheme compiler, the only data point that can be used for comparison between **Blitzkrieg** and Berlin's system is the Runge-Kutta 6 body problem. The speedup of **Blitzkrieg** over Berlin's system is a factor of about sixteen. A factor of four or so is due to the use of floating-point vectors, that leaves another factor of four to account for. Some of this difference is due to different instruction sets, cache sizes, etc. However, **Blitzkrieg**'s extra factor of four in performance comes mainly from the fact that it uses the Scheme compiler, since source-to-source translations are used. Berlin's partial evaluator outputs microcoded primitives in C, which is not nearly as effective. While it may therefore not be so fair to compare **Blitzkrieg** to Berlin's system, the resultant code output by **Blitzkrieg**does appear to have performance comparable if not superior to Berlin's partial evaluator.

3.1.2 Optimization Effects

While running the Solar-system integration code, **Blitzkrieg** took advantage of several simple arithmetic optimizations. These arithmetic optimizations eliminated floating-point operations with a zero as one of the inputs. Specifically, the partial evaluator removed instances of `(flo:+ x 0.0)` or `(flo:+ 0.0 x)` and replaced them with `x`, removed instances of `(flo:- x 0.0)` and replaced them with `x`, removed instances of `(flo:* x 0.0)` or `(flo:* 0.0 x)` and replaced them with `0.0`, and finally removed instances of `(flo:/ 0.0 x)` and replaced them with `0.0`. The effectiveness of these optimizations obviously depends on the availability of opportunities for their utilization in the code. Several measurements were made to examine the effects of these optimizations. Table 3.2 shows the results of these experiments.

It appears from the table that there were some opportunities for arithmetic optimization that **Blitzkrieg** took advantage of. As discussed in the previous chapter, these optimizations were very easy to express. At such a slight cost, further optimizations could be easily employed—it would be fairly trivial for a user to add another arithmetic optimization such as one for multiplying or dividing by `1.0`. It is important to note that the instruction counts displayed in table 3.2 are for single iterations of inner loops.

3.2 Fixed-Format Input Port Example

In order to demonstrate the ease of extensibility of the partial evaluator, I extended **Blitzkrieg** to handle the task of reading data from fixed-format input ports. An input port is a Scheme object used for I/O that serves as a source of input data. Input ports consist of two parts: (1) a means of accessing a data source, and (2) a current position pointer or counter indicating where to get the next datum. One can, for example, use a file as an input port, or even a string. To test the extensibility of **Blitzkrieg**, I applied it to a fixed-format port input application for reading stock market data from a file. This application is representative of many similar data input programs, and provides a good example of how the extensibility of **Blitzkrieg** can be used.

The code to read in stock market data is given below:

```
(define (read-stock-quote)
  (let* ((name (read))
         (high (read))
         (low (read))
         (close (read)))
    (list name high low close)))
```

`read-stock-quote` calls the procedure `read` to input from a port. In the definition above, `read`[2], is a Scheme procedure that returns the next object (such as a symbol or a number) parsable from the current input port and leaves the port pointing to the place in the file immediately following the read object's representation. `read-stock-quote` assumes that each entry in the input port contains four pieces of data about a stock, separated by whitespace: the stock's name, its high price (for the day), its low price (for the day), and its closing price. Thus, by opening an input port to a file, one can read line after line from the file by calling `read-stock-quote` successively.

`read` is a very general procedure that works for reading in all kinds of Scheme objects. Consequently, `read` first parses the data found in the input port in order to determine what kind of a Scheme object is to be read in; then, it dispatches to the input procedure that does the actual work of reading in the appropriate Scheme object. More specifically, `read` looks at the first character from the input port. If it sees whitespace, it discards characters until it ends up with an object; otherwise, it dispatches to the appropriate *parsing primitives*—the procedures `parse-object/symbol`, `parse-object/atom`, or `discard-whitespace`.These input procedures in turn call various *input primitives* to read the object in character by character.

[2]`read` is part of the MIT Scheme runtime library and is fully documented along with ports in [15].

When the format of the input file is unknown, `read` provides a nice general way to read in data from the file. But if the port's format is fixed, calls to `read` result in a lot of wasteful and unnecessary parsing. Such is the case in the stock market example above. Here it is not necessary to parse the port every time a Scheme object is to be read in. Instead, because the object types are known ahead of time, they can be read in directly using the appropriate input primitives.

I now discuss a method to explicitly extend **Blitzkrieg** to handle reading from an input port.

First a port placeholder class, `<input-port-placeholder>` is added to the system:

```
(define-class <input-port-placeholder>
              (<input-port> <placeholder>)
               shadow-placeholder-port)
```

The new class, `<input-port-placeholder>`, inherits from both the `<input-port>` and the `<placeholder>` classes. It also has a slot, `shadow-placeholder-port`, that holds a port shadowing the actual input port represented by the placeholder. This shadowing port contains data representative of the input file format that the input port would hold at run time.

The problem is now to determine how the parsing aspect of `read` can be optimized. There are only a few procedures that do any actual "work" in reading in an object. As mentioned before, a good deal of the time spent by `read` is spent figuring out which primitives to call. For the fixed-format input port application, however, data from the shadow port can be used to determine the format of the input port. Since the format of the input port is known before runtime, the appropriate primitives can be directly residualized. In order to accomplish this, two types of changes need to be made.

First each of the input primitives must be able to handle both input port placeholders and normal input ports. The case for normal input ports simply defaults to the same operation that was done originally. In the case of an input port placeholder, however, the input primitive is called on the input placeholder's shadow port. Thus, calling input primitives on an input port placeholder produces type information from the shadow port, allowing the appropriate input primitives to be called to read in the data. This behavior can be naturally implemented with generic procedures.

Second, the parsing primitives `parse-object/symbol`, `parse-object/atom`, and `discard-whitespace` must be revised to check if the current input port is an input port placeholder; if so, the parsing primitive decides what input primitive(s) to call based on reads from the shadowing port. That is, the parsing primitive calls input primitives on the input port placeholder and uses this information to decide whether to read the next character. It then residulizes a new placeholder, calling the appropriate input primitives on the actual input port associated with the input port placeholder.

The modifications discussed above are all that is necessary to extend the system to partially evaluate calls to `read`. Hence, any input program using `read` from a fixed input format can be similarly optimized. While 45kB of code are necessary to handle reading data input ports, only 2 additional kilobytes of code are required to extend the system to take advantage of partial evaluation. The small amount of revision necessary, even at the very low level of abstraction required for file manipulation, attests both to the good use of abstraction and the ease with which the systems allows extensibility with minimal impact on most user code. Using **Blitzkrieg**, one can easily augment data types with placeholders that mimic their behavior, and then rewrite primitive operations with these objects to allow important optimizations using generic procedures.

After implementing this optimization on our stock file example, I tested the system by performing 2000 reads from several different files. Running this experiment on an HP9000/735 resulted in a speedup factor of 1.35 over the original (compiled) program. The same program run on an HP9000/715 series computer resulted in a factor of 2.5 speedup. File and memory cache seem to play an important role in determining the speedup statistics. Nonetheless, for minimal effort, the partial evaluator was extended to deal with input ports as well as `read`, and I was able to observe some measurable improvement of the program's performance.

3.3 Graphics Transformation Example

I now return to the example of a graphics system in order to show in more detail how **Blitzkrieg** can be extended to perform domain-specific optimizations. The example given here is much like the motivating example from the introduction. It is concerned with computing three-dimensional linear transformations.

In particular, suppose one is interested in partially evaluating the following program that rotates an input vector in three-space:

```
(define (rotate x y z)
  (lambda (vector)
    (matrix*vector (make-x-rotation x)
                   (matrix*vector (make-y-rotation y)
                                  (matrix*vector (make-z-rotation z)
                                                 vector)))))

(define normal-rotate (rotate 45. 30. 60.))
```

The procedure `normal-rotate` takes a vector as argument and rotates it by the specified angles in each of the three dimensions. The inputs to `rotate`, `x`, `y`, and `z`, represent angles of rotation around each of the three axes. Given these angles, the procedures `make-x-rotation`, `make-y-rotation`, and `make-z-rotation` return the appropriate rotation matrices. The details of the implementation of these procedures are not important for this example.

Note that `rotate` contains three calls to `matrix*vector`. For `normal-rotate`, all three rotation matrices are known at compile time. Thus, just as in the motivating example, there is potential for domain-specific optimization: I can use the associative property of matrix multiplication to premultiply the matrices at compile time, leaving only one multiplication of the resultant matrix by an input vector for run time.

Here is the complete specification of this optimization, as given to the system by the user:

```
(define-generic-procedure matrix*vector (matrix vector))

(define-method matrix*vector (matrix vector)
  (normal-matrix*vector matrix vector))

(define-method matrix*vector (matrix (vector <vector-placeholder>))
  (let ((gen (placeholder-generator vector)))
    (if (and (generator-combination? gen)
             (equal? (generator-combination-procedure gen)
                     `(matrix*vector ,system-global-environment)))
        (let ((gen-args (generator-combination-arguments gen)))
          (make-new-placeholder <vector-placeholder>
                                (make-generator-combination
                                        `(matrix*vector ,system-global-environment)
                                        (matrix-multiply matrix (first gen-args))
                                        (second gen-args))))
        (make-new-placeholder <vector-placeholder>
                              (make-generator-combination
                                        `(matrix*vector ,system-global-environment)
                                        matrix
                                        vector)))))
```

Here, `matrix*vector` is declared to be a generic procedure of two arguments, a matrix and a vector. If it is called on known inputs, `matrix*vector` performs the computation using the standard routine for multiplying a matrix by a vector (`normal-matrix*vector`). When the second argument is a vector placeholder[3], the computation is residualized—that is, a new placeholder is made. If the placeholder's generator is a combination obtained by a previous call to `matrix*vector`, then the input matrix is multiplied by the matrix of the previous call to produce a new transformation matrix. Both this new matrix and the original input vector are then put in as arguments of the new placeholder's generator combination. Otherwise, the original input matrix and vector are used to create the generator combination of the new placeholder.

The result of partially evaluating `normal-rotate` is a piece of code that calls `matrix*vector` once rather than the three times required in the absense of our folding optimization. Note that in order to get this optimization to work in our system I had to run dead-code removal in order to get rid of the unused placeholders that were created each time `matrix*vector` was called during the partial evaluation.

The performance measurements made on this program compared the compiled original code with the compiled partially evaluated code. These tests were performed on an HP9000/735, under the same conditions as the ones described in the previous examples. The speedup observed for `normal-rotate` was a factor of 4.04 over the original code. Without having done the above folding optimization, the partial evaluator was able to achieve a factor of 1.32 over the original code. Thus, the domain-specific optimization on this program provided an additional factor of 3.06

[3]As mentioned in chapter two, the class `<vector-placeholder>` is predefined in **Blitzkrieg**. In other situations, the user might have to define his own placeholder class.

speedup. Given the nature of the example, this is not too surprising, as two procedure calls were completely eliminated.

I have shown a simple example of utilizing **Blitzkrieg** to implement a domain-specific optimization on a graphical system application. We have demonstrated that having the information about the program's structure available within the placeholders allows the user to easily manipulate and transform that information. This provides a very powerful mechanism for implementing application-specific optimizations. As the above example demonstrates, having this power can in some cases yield quite substantial speedups of the partially evaluated code.

3.4 Chapter Summary

In this chapter, I showed how the **Blitzkrieg** system was applied to several real-world applications. Its viability was proven by speeding up numerical integrations of the solar system by over two orders of magnitude. Its extensibility was shown on a input port example, yielding a modest improvement in execution speed. Finally the power of **Blitzkrieg**'s ability to allow the user to express domain-specific optimizations was shown on a 3D transformation system, yielding a speedup factor of 4.04 over the original code and a factor of 3.06 over partially evaluated code without the domain-specific optimizations.

Chapter 4

Related Work

Several partial evaluation systems exist for the Scheme language, most notably Berlin's partial evaluator [6], FUSE [23][22], and Similix [7]. In this chapter I compare the capabilities of **Blitzkrieg** with each of these systems.

4.1 Berlin's Partial Evaluator

Throughout this thesis, I have referred to Andy Berlin's partial evaluator from [6]. Berlin's partial evaluator was written as part of MIT's Project for Mathematics and Computation's (Project MaC) scientific computing project at MIT. The **Blitzkrieg** system represents the next generation of partial evaluators from Project MaC. Thus, the two systems are similar in many respects. There are, however, some major differences due to improvements implemented in **Blitzkrieg**.

One of the major advantages of **Blitzkrieg** over Berlin's system is that **Blitzkrieg** is designed to be user extensible. Berlin's system was implemented to work primarily on programs involving floating-point computations, and thus lacked a type system. In **Blitzkrieg**, an extensible typing system is implemented via an object system. This allows the user to create placeholders for data structures unknown to the system. The object system also makes it easy to implement procedures that behave differently for known versus placeholder inputs; in Berlin's system, the user would have to create and use the appropriate predicates to achieve the same functionality. The other advantage of **Blitzkrieg** is that its generic procedures make it easy to modularize code that deals with normal computation versus code that involves manipulation of placeholders. This means that a user need not, in most cases, mix up the normal code with the code for partial evaluation.

I note also that **Blitzkrieg** implements the entire MIT generic arithmetic system, whereas Berlin assumed all generic numeric operations were floating point. In addition, **Blitzkrieg**'s result for floating point code as demonstrated on the solar system integrators are superior to Berlin's results. This is primarily because recent additions to the MIT Scheme system made it possible to use constructs that eliminated floating point `cons`ing. Berlin predicted such results in [6], but did not have this capability when he implemented his system.

Another major difference between **Blitzkrieg** and Berlin's system is that Berlin's placeholders do not carry a representation of the residualized code. Fundamentally, this means that domain-specific optimizations, such as the one shown for the 3D graphics transformation system, would be impossible to accomplish with Berlin's system. This is, of course, because Berlin's system was not designed for user extensibility and domain-specific optimizations. Obviously there is some space cost associated with this since one is essentially carrying a flow graph representation of the program.

4.2 FUSE

FUSE [23] is a set of online partial evaluators implemented at Stanford. FUSE partially evaluates programs written in a functional subset of Scheme.

The FUSE system has a number of advantages over **Blitzkrieg**. One advantage is that FUSE is more sophisticated in dealing with termination and data-dependent computation, essentially because it knows about program points. Code that would cause **Blitzkrieg** to not terminate while being partially evaluated—such as some cases of data-dependent recursion—would be successfully handled by FUSE. FUSE also prevents redundant specialization in order to avoid code bloat, whereas **Blitzkrieg** (as its name suggests) has no such optimizations. **Blitzkrieg** would certainly benefit if these features were added as that would greatly increase the domain of programs it would work on.

On the other hand, **Blitzkrieg** also holds several advantages over FUSE. User extensibility and the ability to express domain-specific optimizations are not design considerations of FUSE. The discussion above contrasting **Blitzkrieg** with Berlin's system with respect to these two points applies here. Although FUSE has a type system, the type system is used to allow users to express information about inputs as well as for the system to propogate and use the type information during specialization. Thus, it is not extensible for the user. The file input example from the previous chapter, cannot be easily accomplished using FUSE by the normal user. Rather, the implementor of FUSE, or person of similar expertise, would have to add this additional capability to the system. Additionally, unlike FUSE, **Blitzkrieg** is not restricted to a functional subset of Scheme. It can handle the many types of mutation that most commonly occur in Scheme code.

4.3 Similix

Danvy and Bondorf's Similix system [7] is a publicly available offline self-applicable partial evaluation system. It also handles only a functional subset of Scheme, with some consideration given to file input and output.

Similix enjoys several advantages over **Blitzkrieg**. Since it is self-applicable, all of the renowned Futamura projections[12] are feasible. In Similix, one can write a so-called "compiler generator" that, given an interpreter, generates a compiler. **Blitzkrieg** is not self-applicable. I note, however, that the pragmatism of self-applicability is unclear if a partial evaluation system is to be applied to most typical user applications, other than compilers and interpreters. Besides being self-applicable, Similix, like Fuse, also has the advantage of possessing better termination properties as compared to **Blitzkrieg**.

There are, however, advantages that **Blitzkrieg** has over Similix. Just as with the other two systems, Similix was not designed for extensibility and the ability to express domain-specific optimizations. In fact, it is not clear how a user could easily express these optimizations in an offline system, particularly if reduce versus residualize decisions are affected by these optimizations. In this aspect at least, online partial evaluation is superior. Again, the discussion above contrasting **Blitzkrieg** with Berlin's system with regard to user-extensibility applies here also. Though Similix has a type system, extensions by the user, such as the file input example, could not be accomplished as easily as they are in **Blitzkrieg**.

In addition, the aggressiveness of offline systems like Similix in optimization is adversely affected by the fact that they are self-applicable. Offline systems decide to reduce or residualize a computation in a phase before the specialization actually occurs. Ruf [22] explains why offline partial evaluators choose to residualize computations that could be computed at run time by an online one. The pragmatism of self-applicability in a partial evaluator is not clear when one considers that a user who uses a partial evaluator probably wants his program to be made as fast as possible.

4.4 Chapter Summary

In this chapter three partial evaluation systems are compared and contrasted with **Blitzkrieg**. Unlike **Blitzkrieg**, none of the other systems give consideration to user extensibility, or allow users to express domain-specific optimizations. On the other hand, **Blitzkrieg** is not as sophisticated as some partial evaluators with respect to the termination characteristics. This, however, was not the major design objective of **Blitzkrieg**. Presumably, **Blitzkrieg** can be easily enhanced address this deficiency.

Chapter 5

Conclusions and Future Work

In this thesis, I described the **Blitzkrieg** user-extensible partial evaluator that works on a large class of MIT Scheme programs. **Blitzkrieg**is based on an object system. Three properties of **Blitzkrieg** and its use of the object system make it easily extensible: First, it is easy to create placeholders that represent types previously unknown to the system. Second, the user can easily make his procedures behave differently for known inputs and placeholder inputs. Finally, in addition to employing standard partial evaluation techniques, **Blitzkrieg** can take advantage of application-specific optimizations based on information supplied by the user. The viability of **Blitzkrieg** is shown by applying it to the Stormer integrator on a 6-body problem (achieving a factor of 610 speedup) and the Runge-Kutta integrator on the same problem (a factor of 365.5 speedup). In addition, the flexibility of the approach is demonstrated by extending the system to handle port input programs, achieving a factor of 1.35 speedup on a representative program. Finally, **Blitzkrieg** is also able to achieve a factor of 4.04 speedup on a graphics application by performing application-specific optimizations.

There is, however, much work left to be done. For example, **Blitzkrieg** has inferior termination properties compared to other partial evaluation systems. This aspect of the system should be improved in future work. Particularly, extending **Blitzkrieg** to handle program points and to reduce redundant specialization would extend the domain of programs that **Blitzkrieg** handles.

Data-dependent mutation is another issue that should be addressed in future work. Most partial evaluation research disregards this very important problem; instead, the researchers concentrate on the functional case. As a result, most partial evaluation systems are inapplicable to typical user programs. A possible approach to solving this problem is to log all mutations during partial evaluation, thus making them reversible. This method would make a comparison of mutated values in different branches of a program feasible. I believe that this is the key to solving the data-dependent mutation problem; while this approach may turn out to be impractical, it deserves further investigation.

There is also some short-term work that should be done. The Scheme back end of **Blitzkrieg** does not generate very optimal code in terms of the amount of storage for intermediate values, since it simply assumes all intermediate values must be stored.

This situation could certainly be improved. Also, because of the difficulty that the MIT Scheme compiler has compiling large basic blocks, work should be done to split up large basic blocks generated, so that the resultant partially evaluated programs can actually be compiled and used.

Future work notwithstanding, I believe that **Blitzkrieg** has shown itself to be capable of speeding up real programs, and that it provides a useful starting point for anyone interested in applying partial evaluation to a wider variety of applications.

Bibliography

[1] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, G. Sussman, "The Supercomputer Toolkit and its Applications," MIT Artificial Intelligence Laboratory Memo 1249, Cambridge, Massachusetts.

[2] S. Adams, personal communications May 10, 1994

[3] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools* Addison Wesley, 1988

[4] A. Berlin and R. Surati, "Partial Evaluation for Scientific Computing," *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, 1994

[5] A. Berlin and D. Weise, "Compiling Scientific Code using Partial Evaluation," *IEEE Computer* December 1990.

[6] A. Berlin, "Partial Evaluation Applied to Numerical Computation", in proceedings of the 1990 ACM Conference on Lisp and Functional Programming. Also see "A Compilation strategy for numerical programs based on partial evaluation," MIT Artificial Intelligence Laboratory Technical Report TR-1144, July, 1989.

[7] A. Bondorf, "Similix 5.0 Users Manaul" GNU Free Software Distribution 1993.

[8] C. Colby and P. Lee, "A Modular Implementation of Partial Evaluation" Technical Report CMU-CS-92-123, School of Computer Science, Carnegie Mellon University, Pittsburgh PA, 1992.

[9] C. Consel and S. C. Koo, "Paramerized Partial Evaluation" *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation*, June 1991, 92-106

[10] O. Danvy, Personal Communication May 1995.

[11] J. Ellis, *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985. Also available as the ACM PhD Dissertation of the Year, MIT Press 1985.

[12] Y. Futamura. "Partial Evaluation of Computation Process— An Approach to a Compiler Compiler" *Systems, Computers, Controls*, 2(5):45-50, 1971.

[13] C. Hanson, "The Scheme Object System Reference Manual"

[14]

[15] C. Hanson, "The MIT Scheme Reference Manual," MIT Artificial Laboratory Technical Report 1281, 1991

[16]

[17] N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generations* Prentice Hall, 1993

[18] S. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison Wesley, 1989.

[19] G. Kiczales, J. Rivieres, and D. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991

[20] H. Masselin, "Efficient Implementation of Fundamental Operating System Services," Columbia University PhD Thesis 1992

[21] G. Rozas, personal communications May 10, 1994

[22] E. Ruf, "Topics in Online Partial Evaluation", Technical Report CSL-TR-93-563, Computer Systems Laboratory, Stanford University, Stanford, CA. 1993.

[23] E. Ruf and D. Weise, "Avoiding Redundant Specialization During Partial Evaluation" In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluationand Semantics-Based Program Manipulation,* New Haven, CN. June 1991.

[24] G. Steele Jr. *Common Lisp: The Language 2nd Edition* Digital Press, 1990

[25] R. Surati, "A Parallelizing Compiler Based on Partial Evaluation", MIT Artificial Intelligence Laboratory Technical Report TR-1377, July 1992

[26] R. Surati and A. Berlin, "Exploiting the Parallelism Exposed By Partial Evaluation" *International Conference on Parallel Architectures and Compilation Techniques*, Elsevier Science, 1994

[27] G. J. Sussman and J. Wisdom, "Numerical Evidence that the Motion of Pluto is Chaotic," *Science*, Volume 241, 22 July 1988.

[28] D. Weise and E. Ruf, "Computing Types During Program Specialization" Technical Report CSL-TR-90-441 Computer Systems Laboratory, Stanford Univerisity, Stanford, CA 1990.