Pythia: A Parallel Compiler for Delirium[‡]

Oliver Sharp[†]

May 22, 1990

Abstract

Pythia is an optimizing compiler for the coordination language Delirium, written in Delirium. It is part of the Madness project, which investigates both an alternative to the traditional dataflow model and an alternative to the traditional dataflow implementation strategy. Delirium embeds imperatively defined operators within a functional context, giving the programmer control over the granularity of a computation. The application's control structure is expressed using powerful functional-language constructs like closures and function valued parameters. The bulk of actual computation, however, is expressed in a convenient imperative language. Pythia performs traditional optimization strategies like macro-expansion, common-subexpression elimination, and constant propagation.

[†] Supported by Hertz Fellowship and Lawrence Livermore National Laboratory

[‡] Preparation of this paper was supported in part by the Defense Advanced Research Project Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292

Contents

1 Introduction

Pythia is an optimizing compiler for the coordination language Delirium, written in Delirium. In addition to serving as a necessary tool, the compiler has two design goals: to demonstrate dataflow graph optimization strategies and to prove that Delirium is able to conveniently express the control structure of a complex parallel program. Parallel tree walking is shown to be a useful methodology for parallel computation which is easily described in Delirium.

Dataflow [1, 2] is an attractive model for parallel computation because it does not introduce artificial sequencing dependencies. A dataflow language compiler translates a program into a graph, where nodes in the graph (called *actors*) represent computations and arcs represent data dependencies. There are two major problems that have prevented dataflow from being accepted: scheduling overhead and data structure decomposition.

Steve Lucco and I worked on a project we called Madness, in which we developed a modified dataflow paradigm that attempts to address both of these difficulties. We were motivated by a desire to give the programmer control over important decisions and to reduce system overhead to the practicable minimum. Section 2 describes the model in detail and section 3 covers Delirium, the functional language on which the system is based. We call Delirium a *coordination* language, because the programmer uses it to coordinate a set of sub-computations rather than to express computation directly.

We are in the process of implementing various applications within our model to test its usefulness. The two most important parts of our development environment are the run time system and the compiler. In our initial work, we developed two compilers and two run time systems; as our ideas were expanded and refined, we found that the original designs were inadequate. The second incarnation of the run time system, which we named Obsession, has proven to be flexible and efficient. It takes advantage of an idea we call *template activation* to achieve excellent performance on a general-purpose shared memory machine. The run time system currently runs on the Sequent Symmetry, the Cray 2, the Cray Y-MP, and the Butterfly TC2000. This paper does not cover the implementation of the run time system, other than to give the performance measurements that affect compiler optimization decisions.

The second version of the compiler, while adequate, was based on a straight-forward translation with a minimum of analysis. We were intrigued by the notion of applying traditional optimization techniques to dataflow graphs, an idea that has been explored by other researchers [19]. Rather than extensively modifying the existing compiler, I decided to rewrite it in Delirium. Pythia is at the same time a useful tool and an interesting application.

Developing a usable compiler in a functional language is something of a challenge, because traditional implementation techniques rely on incremental update of a parse tree that represents the program being compiled. Incremental modification is difficult for functional languages to handle efficiently; existing proposals, like I-structures [3], have not been too successful. Pythia poses the added challenge of parallel execution.

The solution to both problems was the same: base all expensive computations in the compiler on a small set of parallel tree walking primitives. Trees are very tractable structures for parallel processing because they have a regular structure that can be easily decomposed into independent units. However, a parse tree usually has a variety of annotations that create non-tree-like dependencies. If these links were carelessly traversed, two execution threads could come into conflict and corrupt the integrity of the tree. The walking primitives establish rules that prevent such conflicts from happening; any primitive that obeys the restrictions is guaranteed to have deterministic behavior regardless of the number of processors that perform the tree walk.

There are three tree-walking primitives, which are described in detail in Section 6.2. Taken together, they are sufficient for expressing all the traditional parse tree manipulations. Each guarantees an execution order over the tree that allows efficient parallel execution. The most important guiding principle in designing the compiler was to establish and obey the restrictions on the traversal behavior and dependencies of each pass. Having grown accustomed to using the walking primitives, I found each pass relatively straight-forward to implement. The only major change in design from a sequential compiler was the need to split a few passes into a marking and an execution phase.

Our experiences with the compiler (Pythia) and a set of other applications have convinced us that Madness is a useful programming model for a broad range of applications. Without excessive effort, we have been able to write parallel programs that achieve good performance, using programming techniques that are in many ways similar to traditional structured programming.

2 The Model

A traditional dataflow program is converted into a graph, where each node represents a primitive computation. These primitives are generally simple (or *light-weight*) operations like addition and list manipulation. The run time system is responsible for scheduling the primitives when the data upon which they operate becomes available. When an operation that executes for a fraction of a micro-second is being scheduled, clearly the scheduling overhead must be extremely low or the overall performance will be very poor. To address this problem, data-flow researchers have proposed various specialized architectures that handle scheduling in hardware. Even with extensive support, however, a flexible language is difficult to implement efficiently.

One solution has been to use *static scheduling*, a technique for determining the entire primitive execution schedule at compile-time. Unfortunately, this is only feasible if severe restrictions are placed on the underlying language, and has proven useful only in a few specific application domains (notably digital signal processing [13]).

An obvious alternative approach is to put more complicated (or *heavier*) computations in a node. Our idea is to allow the programmer to write primitives in any imperative language (like C or Fortran), as long as the effect of the primitive as a whole

is functional. Within the node, however, the full power of the imperative language is available (with a few exceptions that are explained below). The application consists of a large number of primitives that are invoked by a web of functional control structures. The bulk of a Delirium program is written in the familiar imperative language; we have found that the easiest strategy is to write the entire application that way at first. The careful programmer can then replace the top level of the application with functionally equivalent Delirium code, yielding a relatively painless transition to parallel execution. This is how Pythia was implemented.

Traditional dataflow languages also have difficulty managing large data structures. The normal strategy in imperative languages is to make a series of small (and often related) modifications to various parts of the structure. Current dataflow languages make it awkward to express such updates; even more serious, performance is dismal unless the compiler is clever enough to avoid unnecessary copying (Cann's thesis [4] describes a variety of analysis techniques). By taking advantage of shared memory architectures, we are able to use high-level functions to decompose structures, operate on the pieces, and recombine them efficiently.

Our research into efficient execution of Delirium programs has focused on shared memory multiprocessors. All of the most commonly used supercomputers (like the Cray 2 [5] and Cray Y-MP) have this architecture. We believe that during the next decade personal workstation design will move in the same direction. The relative ease with which shared memory operating systems can be constructed, the utility of such systems in typical workstation tasks (such as the compilation of multiple files), and the large number of shared memory projects all support this conclusion. The latter include commercial machines like the Sequent [8] and workstation research projects like the Spur [11] and the Firefly [20].

Delirium has a significant advantage when executed on supercomputers, because the operators can be expressed in a traditional language to take advantage of vectorizing compiler technology that is already well understood. Furthermore, existing programs written in those imperative languages can be much more quickly parallelized under Delirium than any system that requires programs to be expressed in a new notation.

3 Delirium: The Language

Delirium is a functional language without built-in primitives. The programmer uses it to describe the control structure of the application, taking advantage of powerful constructs like closures and function valued parameters. Actual computation is done by the operators, which can be written in several languages including C, FORTRAN, and compiled Common Lisp.

The earliest dataflow languages were fairly primitive, because the specialized architectures that were designed to support them (like the Manchester machine [10]) could only manage simple tokens. The notion of embedding a graph within a token did not

emerge for some time and is still not embraced by all of the dataflow community. The rest of the functional language community quickly moved to first class functions (described below) and other elegant refinements in languages like ALFL [12].

As mentioned in the introduction, one of the goals behind Madness was to investigate a new execution strategy for dataflow graphs that we felt would be efficient on a general purpose computer. This strategy supports tokens that represent expandable subgraphs, freeing us to include in Delirium features that are common in modern functional languages.

One feature in modern functional languages that we have not incorporated in Delirium is rich support for type declaration and inference – the language is untyped. However, operators can optionally be annotated with type declarations which are enforced by the run time system. We have not used the declarations for our existing applications; type checking proved to be quite unnecessary for the compiler, though it would have identified one or two easily discovered bugs.

Constructs in Delirium are similar in spirit to those found in other languages like ALFL [12] and VAL [15]. The programming style is quite different, however, because of the lack of built-in primitives. The high level control of the application is described in Delirium, providing a way for computation to be spread easily across the processors in a parallel machine. The programmer must be careful to use destructuring operators when dealing with large data structures, or there will be little improvement over sequential code. The design of the compiler itself is one example that demonstrates the kinds of strategies available to the Delirium programmer.

A Delirium program consists of a group of functions, one of them called `main`. Functions are *first class* objects, meaning that they may be passed as arguments, bound to variables, or returned as values (Section 5.1.1 describes functions in detail). Dynamic graphs are important, because they allow a wide variety of control constructs to be expressed compactly. Dataflow languages without support for graph expansion have great difficulty with unpredictable execution strategies like recursion. The Delirium run time system has extensive support for managing dynamic graphs efficiently.

Functions can be applied to arguments and are *curried* if fewer arguments are supplied than were expected. A function of two arguments `a` and `b` can be applied to a single argument `x`. The result is a function of one argument that has the binding of `a` to `x` bound within it. When this new function is applied to an argument `y`, the result is the same as that of applying the original two-argument function to `x` and `y` at once.

Delirium also supplies a macro facility, something used in other languages as a notational convenience or to control evaluation of arguments. In Delirium, macros are intended primarily for convenient operation on large data structures.

3.1 The Basic Language

One advantage of functional languages is their fundamental simplicity. Only a few constructs are necessary to provide a rich programming framework. This section describes the basic structures in Delirium (see Appendix II for a BNF grammar). Each structure is a different type of value-producing expression. A Delirium program consists of a set of functions, one of them called **main**. A function definition consists of a name, an argument list in parentheses, and a body. The body is an expression, which is evaluated when the function is applied. Here is a simple example:

```
main()
  start_things_rolling(42)
```

3.1.1 Atoms

The simplest expressions in Delirium are either constants or binding references. Constant types are integers, strings, and floating point numbers. Some examples:

```
an_integer_operator(34)
a_string_operator(''hello'')
a_float_operator(-345.23)
```

If a variable appears, the name must be bound in the surrounding environment, using standard lexical scope rules.

3.1.2 Multiple Values

Because functional languages eliminate global variables, it is especially useful for them to allow multiple value return. This is even more true in Delirium, which depends on destructuring operators to divide large data structures into many pieces that can be processed in parallel. The syntax is simple:

```
<exp1,exp2,exp3 ...>
```

where the comma separated values may be any valid expression. This package can be passed around as a single value; its components can be separated only within a `let` binding as described below.

3.1.3 Let Bindings

The let binding associates identifiers with expressions. Its syntax is:

```
let
  <var> = <value>
  ...
in <expression>
```

A binding can also be used to subdivide a multiple value:

```
let
  <a,b,c> = op_that_yields_multiple_value(12)
  <d,e> = c
in do_something(a,b,c,d,e)
```

In this example, the variable c must be a multiple value package or an error will be signaled at run time. To manipulate packages, we have provided a set of operators like car, cdr, cadr, etc. These are simple to write and a different set could be defined quickly by a Delirium programmer if desired.

The last kind of let binding creates a function, eliminating the need for a "lambda" construct by making it easy to define functions where they are needed. The syntax is:

```
let
  <name>(<arg>, <arg>, ...) = <body>
  ...
in <expression>
```

Within the expression making up the body of the let, a new function has been defined. Nested bindings are scoped in standard lexical fashion.

Each of the expressions on the right hand side of the equals sign can refer to any of the variables or functions bound in that let statement (some dialects of LISP, like SCHEME [17], call this construct a *letrec*). For example, this is a legal code fragment:

```
let a = salt(15)
    salt(x) = if imdone(x) then 10
                           else salt(next_value(x))
in figure_something_out(a)
```

Of course, if imdone and next_value are not correctly written, this expression may not terminate. A value will be returned if imdone(15) is true or if imdone(exp) is true, where exp consists of some finite number of next_value applications on to the original value of 15.

Mutually recursive functions are also legal, as in:

```
let salt(x) = if imdone(x) then x
                           else pepper(next(x))
    pepper(y) = if hesdone(y) then y
                              else salt(next(y))
in
   salt(give_start_value())
```

3.1.4 Function Application

A function can be applied to a comma-separated list of arguments. This causes the evaluation of the body of the function, replacing uses of the formal parameters with the corresponding actuals supplied by the invocation. As explained above, automatic currying is performed if too few arguments are supplied. Operator invocation has identical syntax and is evaluated by calling the operator on the supplied arguments.

3.1.5 Conditional

The syntax for the conditional is:

```
if <expr1> then <expr2> else <expr3>
```

where <expr1> must evaluate to an integer. As in C, a zero value is false and a non-zero is true.

3.1.6 Iteration

The iterate construct syntax is:

```
iterate
  {
    <iter-variable> = <initial-value-expr>, <update-value-expr>
    ...
  }
while <expression>, result <expression>
```

Evaluation begins by binding each iteration variable to its initial value. If the conditional expression is true, each iteration variable is bound to the value of the update expression. The conditional expression is re-evaluated. The looping ends when the `while` expression is false; the final value of the iterate construct is the expression appearing after `result`.

3.2 Operators

An operator in Delirium is a procedure written in an imperative language like C or Fortran. Each node in the dataflow graph has a corresponding operator, except for a few types that are handled by the runtime environment directly. An example of the latter is a *function application* node, described below in detail. Aside from these special cases, a normal node has some number of inputs, each corresponding to one of the arguments of the function. Some operators return more than one value, in which case the programmer must use a special syntax because languages like C do not support multiple value return.

For the purposes of scope, operators are considered to exist as undeclared top level functions. Normal scope rules apply, so the visibility of an operator named `salt` would be obscured by a let binding with the same name.

There are a few restrictions on operators. They are intended to work functionally, so they are not permitted to use static or global variables. Each time an operator is invoked on the same input, it should return the same output, just as any primitive in a functional language does. This raises a few difficulties — I/O, for example. The traditional solution, in languages like VAL [16], is to this problem is to use *streams*. Input and output values are packaged as a stream of values. The programmer can get the head of a stream (its first value), the tail (the rest of the values), can append streams, etc. We have provided a traditional stream library with Delirium that is designed to interact with the UNIX file system.

Another problem with functional operators is that non-determinism is difficult to manage. We felt that non-deterministic operators might be an interesting experiment, so we allow the programmer to annotate an operator to that effect. This prevents Pythia from changing the number of calls to that operator through optimization. For example, suppose that we define a non-deterministic operator `random` that computes a random number using some external criteria as a seed. This violates the functional model, so it is no longer legal to assume that two seemingly identical applications of `random` will return the same result (and it would be a rather poor random number generator if they always did!). Pythia, warned by the non-deterministic annotation, will not optimize the two calls into a single one as it normally would, routing the value of a single call to both uses. Instead, the output sub-graph would contain two separate nodes representing the different calls (unless either or both were eliminated as dead code — see below).

While operators can be written in any language that can be cross-linked to C, we have provided a pre-processor that makes C operators much more convenient to write. Here is a simple example:

```
float max(val1, val2)
float val1, val2;
{
  if (val1 > val2)
    return(val1);
  else
```

```
    return(val2);
}
```

The preprocessor modifies the code, replacing **return** with code that fills a special token structure with the return value. It also creates a separate file listing all the operators and the types they expect as arguments and return as values. To return multiple values, the programmer lists them as arguments to return, separated by commas. The list of types precedes the function, also separated by commas. For example:

```
STREAM, int check_next(s)
STREAM s;
{
  if (satisfies_test(head(s)))
    return(tail(s),TRUE);
  else
    return(s,FALSE);
}
```

The operator checks to see if some test is satisfied; if so, it returns the rest of the stream and TRUE, indicating that one value of the stream has been consumed. Otherwise it returns the original stream and FALSE. Note that we have provided versions of all the stream primitives that are callable directly by C routines.

Sometimes it is convenient to have an operator that can have different numbers of arguments. In such cases, we supply a set of macro definitions that are more flexible than a static declaration. Here is the last example again, using the C macros for input and output:

```
check_next(s)
{
  STREAM s = GET_INPUT(0,STREAM);

  if (satisfies_test(head(s))) {
    DO_OUTPUT(0,STREAM,tail(s));
    DO_OUTPUT(1,int,TRUE);
  }
  else {
    DO_OUTPUT(0,STREAM,s);
    DO_OUTPUT(1,int,FALSE);
  }
}
```

This version of the macros includes type declarations, which are dynamically checked at run time. A different version dispenses with the declarations, placing that responsibility on the programmer.

One final annotation supported by the pre-processor is **destr**. This notifies the run time system that the specified argument to the function might be destructively modified. Because memory is shared, any number of operators can have read-only access to a data structure through pointers. The run time system keeps a reference count on tokens; any token that enters an operator through a **destr** input will be guaranteed to have no other references. This restriction is enforced by copying the structure when more than one reference currently exists.

Destructive inputs allow very efficient updating of a data structure through side-effects, while still maintaining the purity of the functional model. The programmer who is concerned about efficiency must avoid copying by ensuring that only one copy of the token to be modified will exist when it enters the destructive input. Here is an example of an operator that destructively increments a particular element in a vector:

```
VECTOR *increment_vector(vector, position)
destr VECTOR *vector;
int position;
{
  vector->contents[position] += 1;

  return(vector);
}
```

3.3 Macros

An important goal of the Delirium language is to support convenient ways to manipulate large data structures. The macro facility is helpful, as can be seen from an example. Suppose we wish to perform parallel computations on an array using a fork-join control structure. To avoid repeating the same construct many times, the general idea can be expressed as a macro:

```
macro map(array,op,extra_arg) =
  let <s1,s2,s3,s4> = array_split(array)
  in array_gather(array_apply(op,s1,extra_arg),
                  array_apply(op,s2,extra_arg),
                  array_apply(op,s3,extra_arg),
                  array_apply(op,s4,extra_arg))
```

A macro invocation is replaced by the body of the macro, substituting arguments. The following call:

```
map(an_array,flatten,histogram)
```

has the same effect as:

```
let <s1,s2,s3,s4> = array_split(an_array)
in array_gather(array_apply(flatten,s1,histogram),
                array_apply(flatten,s2,histogram),
                array_apply(flatten,s3,histogram),
                array_apply(flatten,s4,histogram))
```

`map` applies some primitive operator to each element in an array. The programmer has decided that for this application, a good grain size is achieved by dividing the array into four pieces. The operator `array_split` takes an array and returns four commands, each of which is handed to `array_apply`. The second argument is a reference to an operator or function that expects two arguments. `array_apply` walks its piece of the array, applying the operator to each element along with the extra argument (`histogram` in the example). Depending on the operation being performed, `array_apply` or `array_split` might create a second array into which the result of each operation is placed. The four returned values are handed to a combining operator and the final result returned.

Because we are using a shared memory machine, `array_split` and `array_gather` typically need to do no copying. In general, one of the two must allocate a second block of storage that is the same size as the array so that the results can be stored without modifying the original (and thus violating the single-assignment restriction). However, if `array_split` marks its array input as being destructive, and no other copies of the array exist in the system, the extra storage allocation can be avoided. It is up to the programmer to insure that these conditions hold if allocation and copying overhead would be undesirable.

3.4 Example

Here is the entire Delirium code for a simple circuit simulator, demonstrating high-level functions and data structure decomposition:

```
main()
  let
    ntimesteps=50
    init_gates=read_gates()
    wires=read_wires()
  in iterate
       {
         gates=init_gates,map(gates,simulate_gate,wires,i)
         i=0,i+1
       }
     while less_than(i,ntimesteps), result gates
```

The `map` macro is a slight variation on the one given above, in that it takes two extra arguments rather than one. Seven imperative operators are invoked: `read_gates`,

read_wires, array_split, array_apply, array_gather, simulate_gates, and less_than. After reading in the circuit to be simulated, we do fifty steps of iteration.

The work of simulating one circuit element is done by the operator simulate-gate. Here is the C code for it:

```
GATE *
simulate_gate(gatep,wires,timestamp)
destr GATE *gatep;
ARRAY *wires;
int timestamp;
{
  int new_val;

  if (gatep->delay_count > 0)
    --gatep->delay_count;
  else
    (*gatep->output_fn)(gatep,timestamp,wires);

  new_val = (*gatep->compute_fn)(gatep,timestamp,wires);

  if (new_val != gatep->logic_val) {
    if (gatep->delay_count > 0)
      printf("error: logic val change in < 1 prop delay\n");
    else {
      gatep->delay_count = gatep->prop_delay;
      gatep->logic_val = new_val;
    }
  }

  return(gatep);
}
```

The simulation repeatedly computes a new value for the output based on the input, waiting some number of time steps before a changed value is passed onwards because of propagation delay.

4 Programming Methodology

We have found that the principles of designing a Madness program are sometimes quite similar to traditional software engineering techniques. In a C program, a sign of good design is that the main() routine does not do much but call procedures and has few or no global variables. Converting such a program to run under Madness is quite straightforward. Because global variables are outlawed, any function that needs values from the

environment must receive them as arguments and return them when finished. The updated values are then passed to subsequent operations that require them. This sequencing in Delirium code automatically forces the appropriate ordering on the operations.

We have found that there are two common types of parallelism. The first is a group of tasks that can be done at the same time because they do not depend on each other. When the main routine is converted to Delirium, independent tasks immediately parallelize because none depends on the results of the others. If each task is a Delirium function, the various calls are performed in parallel.

The second type of parallelism, much more common, is not so easy to exploit. It has to do with operations performed on data structures. If expressed in traditional imperative programming style, unnecessary dependencies are created that are difficult to find.

4.1 Large Data Structures

One of the most important issues that must be addressed by a parallel programming language is the management of large data structures. In general, a parallel program is much more difficult to write than its sequential counterpart. Most applications that are time-consuming enough to merit such a painful conversion involve large data structures like trees and arrays.

Many languages have found that "apply-to-all" operations make it easy to express complex manipulations of data structures; APL [7] is a notable example in that almost the entire language is made up of a set of operators that perform a regular operation over an entire array at once. Anyone using such a language quickly realizes the opportunity for parallelism that exists in each operation.

An apply-to-all really consists of two parts: an ordered traversal of the data structure and an operation that is to be performed during that traversal. Different computations rely on different aspects of the traversal order. For something like file uncompression, an algorithm might depend on all prior history to decode a particular piece in the middle (though commonly used approaches do not, because of vulnerability to errors). On the other hand, in a cellular automaton simulation, a given cell only depends on the values of its immediate neighbors. The former is difficult to implement in parallel while the latter is simple.

To take an easy case first, imagine that we wish to compute a new array from an existing one by adding three to each element. Rather than doing the addition directly, we could define a function `add_three`:

```
for rows = 1 to ROWNUM
  for cols = 1 to COLNUM
    array2[i,j] = add_three(array1[i,j])
```

The advantage to the reformulation is that it can be used with an apply-to-all oper-

ator.

```
array2 = do_on_each_array_element(array1,add_three)
```

We can define an entire suite of functions like **add_three**, all of which expect to be called on a single element of an array and that return an updated element. When combined with a set of parallel apply-to-all operators, the update functions allow a variety of powerful array manipulations to be expressed compactly and executed efficiently in parallel. The programmer can write functions that expect an element and return an element, unconcerned about whether they will be called during a parallel array walk or a sequential one. The parallel application operators can have different versions, each carefully tuned for a different architecture.

A more complicated walker will handle operators that rely on certain ordering properties. For example, an operator might rely on sequential execution within a given row; the walker can accommodate this by traversing many rows at the same time while still enforcing the constraint that each row be done sequentially.

Going back to the simple example that operates on each array element, there are still two problems. The first is that an apply-to-all is far too inefficient, if every element is done on separate processors, because of the communication overhead. This is solved in Delirium by divide an array into pieces and handling each piece entirely on one processor. The array example earlier in this paper shows how this can be done. The code to invoke the array walker need not concern itself with such details, however, as it is only concerned with the ordering dependencies guaranteed by the walker.

The second problem is that this section has assumed two arrays, where the results of operating on the first are placed into the second. When very large arrays are being used, it can be unacceptable to require that enough storage be allocated for two copies. Furthermore, if only a small amount of data is to be changed, it is absurd to require that the rest be copied. To solve this problem, we have **destr** arguments; they notify the run time system that the same storage will be used for an incoming token as an outgoing one. The semantics from the functional standpoint are that the original data structure is cast aside and a new one returned, but in fact the space is reused. The integrity of the model is maintained because its only requirement is that changes made within one operator cannot affect another. If a data structure that goes into a destructive argument is referenced anywhere else in the dataflow graph, a copy is made.

Our general approach for managing large data structures is to define an operator to divide the structure into appropriately chosen pieces, another to operate on each piece, and a third to merge the pieces. The operator being applied to each element of the structure relies on the ordering dependencies of the process as a whole; a given implementation of a walker can be based on the characteristics of the target machine.

5 Compilation Strategies

Having discussed Madness programming in general terms, I will give a large concrete example by explaining how the compiler is implemented. First, though, it is necessary to understand the transformation that the compiler performs.

The run time system expects a group of function *templates*, each with a corresponding graph. Each of these templates describes a *closure* — a piece of executable code and the environment in which that code is to execute. The code in this case is a dataflow subgraph, and the environment is a set of bindings for the free variables used by the subgraph. The template description includes the subgraph as a list of nodes with arcs between them, the environment bindings needed, and a set of arcs to route constants, function arguments, and environment values. The task of the compiler is to convert a set of functions into a set of templates.

5.1. Basic Compilation

There are five primitive constructs of the language that must be converted appropriately. Once the graph is constructed, it is optimized in various ways. The final result is then output to a file in a special format recognized by the run time system.

5.1.1 Function Definition

As a first approximation, imagine that all function definitions convert to exactly one closure template. If the compiler were to adopt this approach, the output graphs would be rather inefficient due to function management overhead. Later optimization passes, as described in Section 5.2, improve matters by collapsing functions together. For now, however, we will make a few simplifying assumptions.

Functions are normally defined within a let statement. The function may use identifiers that are defined outside of its body; these are called "free variables" and the compiler must ensure that the values are available when the function is applied to some arguments. Because functions can be bound to variables and passed as arguments or return values, the program may invoke a function outside the lexical context where it was defined. The environment variables used in the function's body are no longer present, but they are needed to carry out the evaluation.

The solution is to pass around more than simply the function; the function is packaged together with an environment containing bindings for all the free variables needed. This package of function and environment, a closure, is constructed by a special kind of dataflow graph node called the *closure constructor*. Each function in the program is compiled into a template, the index of which is given to the constructor to put into the closure. In addition, the value of each free variable is passed in. The operator packages this information together and outputs a token containing the closure. Such a token may be passed around the graph just like any other.

Figure 1 demonstrates the creation of a closure for `black`, based on the following code fragment:

```
let basil = 6
    black(x) = some_computation(x,basil,salt,pepper)
in
    blue(black)
```

The identifiers `salt` and `pepper` must be bound within the environment when this fragment is evaluated.

Functions defined at the top level are handled somewhat specially, because they are not created within the framework of an existing function. Each top level function is converted into a template with no external bindings, and any reference to another top-level function is resolved at compile time. The system is not (yet) interactive, so incomplete binding is not permitted.

5.1.2 Function Application

A function application is handled by another special operator called an *expander*. There are two types of invocation; if the called function is known and doesn't need a closure, we embed its template index in the graph and do the call directly. If a closure is needed, the token that represents it is passed to the expander, along with any arguments for the called function. The expander has the effect of replacing itself in the dataflow graph with the expanded graph of the passed-in closure. The template contains information about which of its nodes needs each environment variable and argument; the run time system handles the bookkeeping when an expander is scheduled for execution.

If an insufficient number of arguments are provided, the expander automatically creates a *curried* function. This new function expects the remaining arguments needed by the original one, executing as expected when they are provided.

Figure 2 demonstrates the effect of an expander node when entered by the closure created for `black` in figure 1. In this case, the closure was invoked with the argument 100 as in:

```
let blue(func_arg) = func_arg(100)
in
  <the previous fragment>
```

To evaluate a function invocation, the system must have access to the free variables and the arguments. The former are carried around in the closure, which is created at the time of function definition. Because Delirium is a functional language, environment values can not be changed by execution as they might be in an imperative language. The arguments are supplied to the expander node at the time of invocation, as shown in the figure.

5.1.3 Let Bindings

Each variable binding in a `let` generates a subgraph corresponding to the expression on the right hand side of the equals. Each use of a let-bound variable represents another arc from that subgraph. The subgraph result may be sent to an arbitrary number of target nodes within the function template.

Figure 3 shows the dataflow graph that corresponds to the following code fragment:

```
let red = <expr1>
    green = <expr2>
    purple = <expr3>
in
    some_operator(14,red,green,purple)
```

Multiple variable bindings are a little more difficult:

```
let <red,blue> = operator(x)
in someother_function(red,blue)
```

Delirium semantics requires that `operator` return a multiple value package with two elements. Such packages are first class, so if `x` was defined elsewhere to be a two element package `operator` could legally pass it through. The compiler looks up the definition for `operator` to decide how to convert the construct. If `operator` returns only one value, that is assumed to be a package and is automatically routed to a decomposition operator with two outputs. If `operator` returns two values, they are used directly. Any other number of outputs is an error.

If an operator that returns multiple values is ever used outside of a multiple value binding, its outputs are automatically routed to a package constructor and the package is considered to be the return value.

5.1.4 Conditionals

There are also two control constructs, the first of which are conditionals. Compilation of conditional expressions is a bit complicated. There are two main sources of difficulty: the dataflow model used by Madness and a desire to avoid unneeded computation. Graphs are executed very quickly by the run time environment due to a strategy we call "template activation," which is detailed elsewhere [14]. This strategy is based on the idea that a node fires when all of its arguments have arrived. The other complication is that one of the two clauses of a conditional will not be needed. In our experience, it is almost universally true that the test clause requires much less computation than the two result clauses. We therefore decided to evaluate the test clause first, followed by only the appropriate result clause. Obviously we could evaluate both and choose the result at the

end, but we felt a more efficient policy was worth the extra work necessary to implement it.

We had to add one departure from the pure dataflow model to accomplish our goal: *null tokens*. These are tokens that do not have any value and are used simply as placeholders to cause a node to fire. The simplest way to use null tokens would be to let them cascade through the graph, handled as a special case by the run time system to avoid any operator execution. We decided to improve on that by introducing null token *forwarding* as well. Any arc in the graph can be annotated with a forwarding address; when the run time system is about to output a null token onto an arc with forwarding, the token "jumps" directly to the specified arc. With this facility in place, we can now compile conditionals efficiently.

Because we are dealing with a dataflow graph, any node with only constant inputs or with none is scheduled for execution as soon as a template is expanded. We would have needed major changes to our model to prevent that from happening. However, we suspect that most long computations will require input from the environment, and these we can prevent from executing by introducing a new kind of special node we call a *gate*. There is one gate each for the true and false clauses, and any variable binding used by either is routed through the gate. Each gate also has an input from the test expression; the gate favored by the test (the true gate if the test is true, the false gate otherwise) outputs the values of all the environment bindings routed through the gate. The other gate outputs a null token along the first output arc.

The first output arc of each gate is set up to forward null tokens to the appropriate input of another special kind of node called a *collector*. The collector has two inputs, passing through the non-null token.

While this strategy does not totally eliminate unnecessary computation, we are confident that most cases will be caught. A careful programmer can ensure that a particularly expensive computation will not be unnecessarily computed by referencing any environment variable.

The strategy is most easily understand with a graphical example. Figure 4 shows the graph that corresponds to the following code fragment:

```
let x = 12
in
  if iseven(x) then <true expr>
               else <false expr>
```

The fact that `x` is passed to the expression graphs through the gates indicates that both expressions use the binding.

5.1.5 Iteration

Traditional iteration does not fit within a functional context. Delirium uses the same solution that was adopted by dataflow languages like SISAL [16], where each "iteration" is a function call. The value of the iteration variables can be updated each pass through the loop without violating the functional model. The iteration function is always tail recursive, so the compiler will mark the call appropriately. Tail recursive calls are handled very efficiently by the run time system, costing little more than a normal node scheduling. Scheme [17] uses the same approach to handle iteration. Here is an example of the conversion:

```
iterate {
    x = "alpha", succ(x)
    y = "joe", parent(y)
}
while not_done(x,y), return y
```

is converted to:

```
let iterator(x,y) =
      if not_done(x,y) then iterator(succ(x),parent(y))
                       else y
in
  iterator("alpha","joe")
```

5.2 Optimization

One of the main goals of Pythia was to explore the optimization of dataflow graphs using standard compiler techniques. Out of the many candidates for inclusion, I chose four that seemed particularly applicable. There are others that might have been worth including, such as strength reduction, but they require a great deal more analysis of the operators than is currently feasible. The pre-processor would need to be much more sophisticated if such optimizations are to be done.

The use of standard optimizations on dataflow graphs has been discussed in the literature [19] [18]. This seems like a useful area for further research, as the future of functional languages depends heavily on improved compilation strategies.

The following sections describe each of the optimizations performed by the compiler. Section 6.3 gives a detailed explanation of the implementation.

5.2.1 Inline Expansion

A function application involves a fair amount of machinery to create and expand the corresponding closure, so it is often more efficient to expand small functions in place.

To make a closure, the environment bindings must be passed to a constructor node and the output token then given to an expander. Closure construction is quite fast, but expansion involves more overhead (see Section 7 for the costs of various run time system actions).

Some programmers use a locally defined function as a notational convenience, equivalent to a local lambda in LISP. There can be many such functions and it would be inefficient to convert each into a closure. The compiler uses a simple heuristic to decide whether to expand a call. If the function requires a closure, it will have been marked in a previous analysis pass and no expansion is done. Only relatively small functions are expanded, to prevent exponential growth of a call-intensive program. The cut-off parameter is tunable through a command line argument; after some experimentation, we have settled on a 10 node subtree as the largest function to expand and that seems to work well. We expect to try a range of values as we move to other large applications with differently structured Delirium source. The number may be somewhat architecture dependent as well, though this has not been true on our existing platforms.

To gain the maximum benefit from inline expansion, Pythia does it repeatedly until fixpoint (i.e. the pass executes without making any changes). The general idea is that call sites suitable for expansion are marked in a first pass over the tree. Two expansions must not come into conflict with each other, so Pythia does not expand a call if there is any possibility that a call site within the called function will also be expanded during this pass. Avoiding the problem requires a recursive traversal of the called function.

After all sites are marked, an expansion pass replaces each marked call with the body of the function. To avoid scope difficulties, a `let` is wrapped around the copied body with appropriate renaming. Here is a fragment that demonstrates the problem:

```
let misc(y) =
      let x = 10
      in some_func(y)
    x = 20
in misc(x)
```

After the straight-forward expansion, we would get the following:

```
let x=20
in let x=10
   in some_func(x)
```

Notice that `some_func` is called with the value 10 instead of the original 20. The difficulty is caused by the let binding within the body of `misc` that has the same name as the actual parameter in the call. The solution to the problem is to do a systematic renaming of the formal parameters. Each of the argument expressions is bound to the new name and references are modified during the copying. A correct expansion of the last example would be something like this:

```
let x = 20
in let y_1234 = x
   in let x = 10
      in some_func(y_1234)
```

The renaming guarantees that some_func will see the correct value regardless of scope problems. The extra level of indirection introduced by the let has no effect on the number of nodes in the final graph, because variable let-bindings are collapsed during compilation without adding any overhead.

5.2.2 Common Sub-Expression Elimination (CSE)

CSE is particularly useful and straight-forward in a functional language because no side-effect analysis need be done. It is somewhat less useful in Delirium, which is used to express control structure rather than computation and is less likely to contain many suitable sub-expressions. However, I decided that CSE was worth implementing because it is a traditional optimization and could be useful if numerous symbolic constants were used. The process is the same for Delirium as for any other functional language – if two expressions look the same (after substitution for free variables), they are guaranteed to evaluate to the same result. The only exception is non-deterministic operators, which prevent CSE from being done.

Notice that CSE corresponds to finding the common expressions and binding them in a let. For example, the following code fragment:

```
let x = compute1(update(john))
    y = compute2(update(john))
in
  compute3(x,y);
```

can also be coded this way:

```
let temp = update(john)
    x = compute1(temp)
    y = compute2(temp)
in
  compute3(x,y)
```

5.2.3 Constant Propagation

Delirium constants include integers, reals, and strings. When they appear, the compiler can sometimes trim the size of the graph by propagating them through bindings and applications.

5.2.4 Dead Code Elimination

Code that is "dead" (i.e. unused) is easily identified in dataflow graphs because by definition there is no path from the dead node to the function's return node. This could happen if a let binding is created but not used in the body, for example. In terms of traditional flow graphs, a node will be pruned if its result does not reach the return node.

Pythia *will* eliminate calls to non-deterministic operators that do not contribute to the final value, so side-effecting dummy calls must pass their return values to some expression that affects the function's return value or those calls will be removed.

6 The Compiler Design

The compiler is conventional in basic design, involving multiple passes over an abstract syntax tree. However, incremental updates to a large tree are not natural operations to perform in a functional language. Imagine a compiler optimization like inline expansion. To decide whether a given function call should be expanded, the compiler needs to know how large the invoked procedure is, what external variables it needs, and perhaps how many other call sites exist. A different optimization will need other information.

If a pass is to be implemented by walking the tree, the walk must be done in the right order so that any necessary external information is computed before-hand. Sometimes there are ordering constraints on the transformations, while other times all the nodes in the tree can conceptually be updated in parallel.

6.1 The Tree Data Structure

For apply-to-all operations to be generally useful, they must manipulate a data representation that is flexible enough to handle a broad range of applications. I chose to use the following C structure to represent a tree node:

```
typdef struct tree_node {
  int node_type;
  int weight;
  int pass_weight;
  char *description;
  int number_of_children;
  struct tree_node *children[];
  char *extra_info;
} NODE;
```

Every node is identified to be of some particular type, so the tree walking routines can invoke their client functions on only particular kinds of nodes. To use a node, the

`description` pointer is cast into the appropriate type of structure pointer, based on the node type.

I chose to have an array of child nodes, rather than a linked list, because the compiler commonly accesses children in an unpredictable order. It is much more efficient to allow direct addressing.

The `weight` entry is an approximation. The routines that decompose a tree for parallel processing use this number to balance the computational load. Sometimes the expected cost of a computation for a particular subtree has little to do with its size. For example, during inline expansion we may be handling only a few call sites scattered over the tree, each of which represents a lot of work (copying the body of the callee). To improve balance in these cases, the data structure has a field called `pass_weight`. This field can be set by an earlier pass to accurately reflect the cost of handling a subtree, without affecting subsequent passes for which subtree size is a good predictor for execution time. As was explained above, inline expansion passes consist of two steps. The first, which marks the application nodes suitable for expansion, also sets the `pass_weight` field to reflect the real amount of work to be done.

6.2 Tree Walking Operators

We need to process the tree structure in parallel, but the ordering dependencies are different in the various passes, so I designed three tree walking primitives. These traverse the tree, applying a given operator at each applicable node. They are implemented by dividing the tree into several subtrees, some set of which is handed to each application operator. Once each piece is processed, they are merged again and the resulting (modified) tree is returned. The tree input is marked `destr`, because of the destructive changes, but the compiler is quite careful to ensure that there is only one copy of the tree in existence when it enters a walker.

- top-down update — walk the tree, updating each node as it is encountered. An update can rely on updates having been completed beforehand for all of its ancestors. The operator is given the node to update and an extra argument that is the same for the whole walk.
- inherited-attribute update — walk the tree, computing an inherited attribute as the traverse moves down. For each node, hand the operator an information package that represents all computations on the way down.
- synthesized-attribute update — walk the tree from the bottom up, doing an update of a given node based on the information that has been computed for each of its children.

These functions are macros; they expand into a block of code that makes calls to special operators that divide, traverse, and recombine the parse tree. To show in more

detail how a walk is implemented, I will explain in detail how the tree is divided during a top-down update pass.

At the top level, we have the Delirium code that expresses the control structure, a simple fork-join:

```
macro tree_walk_update(tree,node_type,operator,extra)
  let <ut1,ut2,ut3> = tree_up_chop(tree,node_type,operator,extra)
  in tree_merge(tree_up_op(ut1,node_type,operator,extra),
                tree_up_op(ut2,node_type,operator,extra),
                tree_up_op(ut3,node_type,operator,extra))
```

This macro is given four arguments: the tree to operate on, the type of node this pass is applicable to, an operator that updates a tree node appropriately, and an extra argument that the operator may need. Here is pseudo-code for a sequential implementation of the walker:

```
update_walk(tree,node_type,operator,extra)

  if the tree node is of type node_type
    call operator(tree,extra)

  for all children
    call update_walk(child,node_type,operator,extra)
```

Note that we do not guarantee any ordering on traversals of the child sub-trees. All ancestors of a node are guaranteed to be seen before the node, but siblings can be seen in any order. This is the crucial property that allows updates to be done in parallel. Here is the pseudo-code for the tree_up_chop routine:

```
/*  tree_up_chop - walk the tree, constructing FAN_OUT sets of
 * sub-trees, each having approximately the same total weight.
 */

tree_up_chop()
{
  node_type = get input telling us what kinds of nodes to operate on
  tree = get input giving tree
  op = the operator we're applying during this walk

  make an array of tree packages
  target = the total weight of the tree divided by the number of pieces
  call recursive_allocate(tree,package_array,node_type,op,goal)
  output packages
```

```
}

recursive_allocate(tree,package_array,node_type,op,target)
{
  if this tree node's type == node_type
     call op(tree)

  for i = 0 to number of children of the tree node
     if child's weight is at least 1/3 of the target weight
        put the child in the package that has the least total weight
        update the chosen package's weight
     else
        recursive_allocate(the child,package_array,node_type,op,goal)
}
```

To ensure that the sets of subtrees allocated to each processor are roughly equivalent in weight, every tree node is annotated with the size of the subtree below it. We divide the total weight of the tree by the number of processors we will be using. The tree traversal runs until we find a subtree that is less than one-third of the desired weight.

After each of the sets of subtrees has been similarly handled, they are merged into a single tree again. In the case of the top-down update walk, there is no work left to perform so the merge simply returns a pointer to the entire tree. The synthesized attribute walk, on the other hand, must run over the crown of the tree finishing the pass now that the values for the subtrees have been computed. Here is the pseudo code for that routine:

```
syn_merge(op,extra,package_array)
{
  op = get input giving op to work on
  extra = get input with extra info
  package_array = get all the packages

  get the original root from one of the packages
  value = syn_walk(root,op,extra,package_array)

  output the root
  output value
}

/*  syn_walk - walk the tree, computing a synthesized attribute.  When
 * you encounter a node that has been spawned off, use the result value
 * from the package.
 */
syn_walk(tree,op,extra,package_array)
```

```
{
  if tree was clipped by syn_chop
    return attribute computed for it (which is in one of the packages)
  else if tree has no children
    return op(tree,NULL,extra)  (the 2nd arg is info from children)
  else
    make an info array large enough to hold the value from each child
    for i = 0 to num_children-1
      info[i] = syn_walk(i'th child,op,extra,package_array)
    return(op(tree,info,extra))
}
```

The walk does a normal synthesized attribute computation, except that subtrees that were clipped are not traversed. The work for that subtree has already been done in syn_op and the attribute can be taken out of the appropriate package.

Notice that we are destructively modifying the tree, so we mark the inputs appropriately; the system will ensure that structures are copied if necessary. The compiler ensures that no such copying is done by having each destructive pass operate on the result of an earlier one. Sequential dependencies ensure the proper sequencing and prevent copying.

6.3 Compiler Passes

The compiler consists of several passes. The following sections describe each one in detail, sketching the algorithms used to implement them. The complete Delirium code for the compiler appears in Appendix III.

6.3.1 Building the Tree

The first step in compilation is to build the parse tree. I decided to use LEX and YACC originally, expecting to recode the passes in parallel if they proved to be a bottleneck. Various research projects have investigated parallel lexing [21] and parsing [9] [22] with varying degrees of sophistication. It proved to be unnecessary to adopt a clever solution, because YACC is able to run in parallel under Delirium as long as the C compiler on the target machine allows global variables to be placed in unshared storage. Each processor runs a self-contained YACC; the subtrees are merged into a single parse tree at the end. The process is as follows:

1. Lex the source code into an array of tokens.
2. Do a fast, dumb recursive descent parse based on delimiter matching to divide the source into top level functions.
3. Allocate contiguous ranges of top level functions to a set of buckets.

4. Hand each bucket to YACC to get a parse tree for that fraction of the code.

5. Generate an overall parse tree from the partial trees.

This yielded very good load balance (as will be discussed below) for parsing, and was much simpler than any complex parallel parsing strategy. I decided that lexing represented such a small fraction of the compilation time that I could ignore it.

The only thing interesting about the conversion of source to parse tree is that the tree is built out of the tree nodes described above. This lets generalized walkers process the tree without understanding its structure. Macro definitions and calls are collected but not yet expanded; that is taken care of in the next pass. The result of this phase is a largely unannotated parse tree; only the bare textual information is represented, without any analysis. References are unresolved, free variables uncollected, and so forth.

6.3.2 Macro Expansion

The next step is to expand all macro invocations. Currently, the semantics are to do a straight replacement with no renaming. Every use of a macro argument in its body is replaced by the corresponding actual parameter expression. If scope conflicts prove to be a nuisance, macro expansion can be made similar to inline expansion of function calls.

The expansion pass is run repeatedly until fixpoint, to allow macros to call other macros. The pass is done with a synthesized attribute walk that computes a boolean that reports whether any expansions were performed.

6.3.3 Environment Analysis

The environment is analyzed in several passes. The final goal is to annotate each use of a variable with the corresponding definition site and to mark functions that require closures. This second task is not as easy as it might seem. The rule for closures is simply that all functions must be closed that depend on an environment or are passed as an argument or return value. The difficulty arises in mutual recursion. Suppose we have the following Delirium code:

```
let one(x) = something(two(x))
    two(y) = if pred(y)
                then one(y)
                else y
in one(1)
```

During the reference reconciliation phase, the compiler will mark one as having the environment variable two and vice versa. Neither function really needs a closure, however, because one and two are known to each other directly. To avoid creating unnecessary closures, the compiler iterates a closure marking pass to fixpoint.

Gather Free Walk the tree bottom up, gathering references to identifiers. When bindings are encountered (like let bindings or function arguments), eliminate them from the list that is passed upwards. Annotate each function definition with the free variables referenced within it. The same thing happens at conditionals, to set up the gates that were described in Section 3.1.5.

Gather Free is not very intelligent. It thinks that functions have many more unbound variables than they do because it doesn't know about operators or unclosed functions. The reason it can't be given that information is that scope rules may hide definitions from the bottom of the tree.

Reconcile References To resolve references, each variable reference is updated with a pointer to its binding site (or *def site*). There are six types of bindings: operator definitions, let-bound variables, let-bound functions, function arguments, conditional shadow variables, and environment variables. The last two of the six are not "real" def sites; they are convenient fictions for the compiler and contain pointers to some other def site. By following a chain of shadow variable and environment variable pointers, one will always eventually end up at one of the first four kinds of bindings.

Reconciliation is handled by an inherited tree walk. The operator is given a stack that represents the set of identifiers with known bindings. When a use is encountered, the operator does a lookup and annotates the use with its definition site. The stack semantics of the lookup ensures that the lexically nearest definition will be found first.

When new bindings are encountered, they are added to the stack and passed down to the children. At a let statement, add all the bound names to the stack for each child. If we have a function definition, also add the arguments during the traversal of the corresponding function body.

Mark Extra Because the Gather Free pass blindly made shadow and environment variables for everything, we have many extra definitions that should be eliminated. Mark Extra is a top-down update walk that marks unnecessary definition sites that can be eliminated.

As was explained previously, it is not immediately apparent whether a given function should be closed or not. Mark Extra must make a decision, because closed functions are invoked via their closures that are shadowed or put into the environment when necessary. The third argument to Mark Extra tells it how to handle references to functions. The options are: always assume every function is closed, always assume every function is not closed, and look at the `needs_closure` field of the function.

Mark Extra is called three times during analysis, once with each of those arguments. The first time, the conservative assumption gets rid of all the easily prunable cases. The second pass is optimistic, mistakenly marking many required definition sites as unnecessary. After Mark Closures uses this optimistic information to mark every closure properly, the final call to Mark Extra has the information it needs to do correct and

reliable annotations.

Prune Extra Once a set of definition sites have been marked unnecessary, the pruning phase eliminates them. This is a simple matter of removing the site from the linked list in either a conditional gate or a function definition.

The third argument to the pruning operator tells it whether functions are properly marked with the `needs_closure` flag. If so, references to unclosed functions are pruned correctly.

Mark Closures The algorithm for determining whether a function needs a closure requires iteration to fixpoint because it begins with the optimistic and usually incorrect assumption that all functions are unclosed. Each pass examines the free variables of every function that is marked unclosed. If any of the variables refers to anything other than an unclosed function, the `needs_closure` flag is set. Any function that calls a newly marked function will be picked up on the subsequent pass.

The pass is a synthesized attribute walk that reports whether any changes were made. The iteration continues until the return value is false.

6.3.4 In-Place Expansion

The expansion pass iterates to fixpoint. Each pass is two-part – a marking phase and an expansion phase. As explained previously, this distinction is necessary to avoid clashing between two expansions during parallel execution.

The marking phase examines each call site in the graph. A called function may not be known — the called function might, for example, be a variable that was passed into the current function as an argument. Expansion is only done for known functions that are smaller than some maximum size and that do not require closures. To decide whether such a call should be expanded, the compiler traverses the body of the called function looking at all of its call sites. If any of those would cause an expansion, the original call site is left unmarked.

When exploring the body of the called function, each of its call sites must in turn be traversed recursively. This exploration could continue infinitely for recursive or mutually recursive functions. In a large call graph, it might also be very time consuming. The first problem could be solved by marking functions as they are encountered, but this is illegal in Delirium because the operator is not permitted to write into those other parts of the tree. Furthermore, it could easily be true that the same function was called from different places; two marking traversals could interfere with each other and get incorrect results.

The solution I adopted is to place a depth bound on the recursive walk. This prevents both infinite walks and overly lengthy ones. Here is an example:

```
let a(x) = a_function(x)
    b(x) = compute(a(x))
in b(some_value)
```

The invocation of b causes the compiler to examine the body of b. This recursively involves the traversal of a and compute. Suppose the latter requires a closure, so it is not a candidate for expansion. The algorithm will mark the original call to b for expansion if and only if it is sure that the call to a_function will not be expanded. Suppose that a_function has a deep call graph that is not fully explored within the depth bound. In that case, the walk would report that it isn't sure what will happen. However, when the compiler encounters the call to a_function directly, it will go slightly deeper into the call tree because the depth bound will not have been decremented twice. The extra search may reveal that the expansion can be performed safely. To avoid problems, a call to a function containing a possible expansion is never expanded.

On the other hand, suppose the compiler can determine that calls to both a_function and compute will not expand. In that case, on the first expansion pass the call within the body of b will be expanded. During the second pass, the call to b also expands, yielding this final result:

```
compute(a_function(some_value))
```

It is obviously much more efficient to evaluate this expression than the original one, which involved the creation of two closures and the machinery to invoke them.

6.3.5 Constant Propagation

Every node that represents the use of a variable is examined. If the variable is bound to a constant in a let construct (either directly or through a definition site chain), the variable use is replaced by the constant. The pass is a synthesized attribute walk that returns a boolean indicating whether any changes were made. The constant propagation operator either leaves the existing node, if the binding is not a known constant, or replaces that node with a constant node.

The compiler iterates the propagation until all replacements have been pushed through. Currently inter-function constant propagation is done only for external bindings. It might be interesting to push constants through calls, when possible, but I tend to doubt whether it would be much of a gain.

6.3.6 Mark Used

After inline expansion, some function definitions are no longer used. The compiler simply marks them unused but leaves them in the tree, primarily for debugging purposes. No further computation is performed on the unused functions.

6.3.7 Common Sub-Expression Elimination

CSE is performed on every function definition by a top-down updating walk. Each of the functions that are left in the tree at this stage in compilation correspond to a closure template in the output, so inter-function CSE is not possible (and I doubt it would be useful in any case). The operator walks the function's subtree in bottom-up order, hashing the growing subexpressions into a table. If two entries coincide, the compiler checks to see whether the corresponding sub-expressions are identical. When they are, the compiler annotates one as being identical to the other. At graph conversion, one of the subexpressions will generate graph nodes and the value will be sent to both uses.

Non-deterministic operators prevent CSE from being performed. As mentioned previously, the compiler guarantees the the number of invocations of such operators will not be changed by optimization. Any sub-expressions that contain one are simply ignored by this pass.

6.3.8 Graph Generation

Each function in the tree is converted into a closure subgraph; the subgraph is expressed within the tree as an annotation to the function definition. The conversion process traverses the tree, annotating each tree node with a graph node that outputs the value for the subtree rooted there. For a detailed understanding of the conversion process, see the (heavily commented) code.

6.3.9 Dead Code Elimination

An update walk is done on the tree, applying the pruning operator to the subgraph at each function definition. The traversal is a recursive walk that marks each node with the boolean `reaches_return`. When a node is encountered, the compiler looks at every node it outputs to. If one of them is the `return_node` or has its `reaches_return` field set, the boolean is set TRUE. Otherwise every node is marked as having been seen and the recursive walk is performed on each target. If any of them reaches the return node, the boolean is set for the current node. Every node that does not have the boolean set is pruned out of the graph.

6.3.10 Graph Output

Output is done by a sequential operator. It outputs a set of templates, each representing a closure (the BNF for the intermediate form is given in Appendix IV). A template consists of a set of nodes with arcs between them, along with three special kinds of arcs. The special arcs correspond to function arguments, environment variables, and constants. Constants are represented directly so that the run time system can handle them efficiently.

Given the following simple Delirium program:

```
main()
  let a = 10
      b = f(a)
  in operator(b)
```

where f and operator are both sequential operators, here is the corresponding template description:

```
ntemplates: 1 start_template: 0
template index: 0 n_env: 0 n_const: 1 n_args: 0 n_arcs: 3
n_nodes: 3

arc num 0 to 1
arc is word: 1
arc is destr: 0
fwd null: -1 -1
constant type: 0 data: 10

node 0 intr: 0 type: 1 template: 0 name: return_node
noutputs: 0

node 1 intr: 0 type: 0 template: 0 name: f
noutputs: 1
ncopies: 1
arc num 0 to 2
arc is word: 1
arc is destr: 0
fwd null: -1 -1

node 2 intr: 0 type: 0 template: 0 name: operator
noutputs: 1
ncopies: 1
arc num 0 to 0
arc is word: 1
arc is destr: 0
fwd null: -1 -1
```

7 Performance

The performance of the compiler can be analyzed from two perspectives. On the one hand, Pythia is a parallel application, running under the Delirium run time system. On the other, it is an optimizing compiler that can be evaluated by the quality of its output.

7.1 Pythia as Application

The first time we ran the compiler in parallel, speedup figures were disappointing. The run time system has a timing facility that allows the Delirium programmer to profile an application. Looking at the costs of executing each operator, there were two obvious problems. In some cases, the load was not very well balanced between the parallel computations. The other problem was sequentially executing operators that took longer than expected. After a few days of modifications, the figures were much improved – running on the Sequent Symmetry, we achieved a 1.5 speedup with two processors, 2.1 with three. Speedup is expressed relative to the sequential version of the compiler.

The sequentially executing parts of the compiler made up roughly one-quarter to one-third of its run time. The remainder parallelized well; parallel tree walking proved to be highly effective in decomposing the load across processors. Here, for example, are the timings of each operator during a synthesized attribute walk applying `mark_2_expand` (in microseconds):

```
call of tree_syn_chop took 2224
call of tree_syn_op took 10888
call of tree_syn_op took 13533
call of tree_syn_op took 13759
call of tree_syn_merge took 3295
```

Notice that the load balance is nearly even, and that tree division and merging are considerably cheaper than the computations. Approximately half of the time spend in `tree_syn_merge` involves useful work that must be done in any case. The rest of the merge, as well as the time needed to chop the tree, represent overhead imposed by walking the tree in parallel rather than sequentially. Each operator also reads inputs and generates outputs, adding another millisecond or so of overhead. In total, the parallel version is roughly 10 percent more expensive. The speedup for this particularly pass executing on three processors vs. the original sequential code is 2.

7.2 Graph Optimization

There are two kinds of nodes in the graphs produced by Pythia: normal nodes that correspond to user-defined operators and extra nodes that manage control. The latter include conditional gates, expanders, closure constructers, and so forth. Pythia is not involved in the compilation of user operators, so the optimizations are all designed to reduce the number of extra nodes that are added. The optimal output graph would only have nodes corresponding to user-defined operators.

The most expensive overhead introduced by the compiler involves the creation and invocation of closures. Table 1 shows the costs of various actions handled by the run time system while executing on a Sequent Symmetry.

operation	*time (in μsec)*
Node Scheduling	150
Closure Allocation	1500
Closure Fill	1500
Empty Closure Invocation	1500
Full Closure Invocation	2500
Cached Closure Invocation	600
Tail Recursive Invocation	400

Table 1: Run Time System Overhead

The numbers are averages, based on the microsecond clock timings we have gathered for a variety of applications. The first field shows the time to schedule an ordinary node with two inputs and two outputs. The other fields break down the cost of managing closures.

A Delirium function call corresponds to the dynamic expansion of a graph at run time via an expander node. There are two ways an expander node can acquire the graph: a token representing the graph can be passed in as an argument, or a direct reference to the target graph can be hard-wired into the node at compiler time. The latter approach only works when a known environment-less function is being called.

In the general case, where a closure token is to be passed to an expander node, the token must be allocated, filled, and then the closure invoked. The entire process requires roughly 5500 microseconds for a large closure. A subsequent invocation of the same closure would only require 600 microseconds because of run time system caching – the 1500 microsecond invocation creates a set of token buffers that can be reused after the closure returns. One such set must exist for each instance of the closure that is simultaneously active.

A direct invocation of the closure eliminates the first two steps and improves the efficiency of the third. The first invocation requires only 1500 microseconds; as in the general case, an invocation that uses a cached buffer set requires roughly 600 microseconds.

Tail recursive calls reuse the current buffer set, yielding slightly better performance than a cached call.

Because closure manipulation is easily the most expensive operation performed by the run time system, compiler optimizations that reduce or eliminate the extra closure management nodes have the greatest effect on execution time. The most effective optimization is inline expansion, which eliminates the machinery entirely. A 5500 microsecond operation is replaced by one that requires only 150.

Another important optimization is closure analysis; by realizing that mutually-recursive routines do not have environment variables, the compiler can use the cheaper alternate

calling strategy. Constant propagation helps here as well by eliminating unnecessary references to variables bound outside the function.

Unfortunately, none of the real Delirium applications we have written gain much benefit from graph optimization. The compiler, for example, is extremely coarse-grained; compilation of the control structure for a medium sized applications takes on the order of 2.99 seconds on one processor. Of that, 2.79 seconds is spent inside user operators doing useful work. The remaining .2 seconds is overhead – roughly 6.5 percent. When optimization is turned off, overhead is increased by 30 milliseconds, adding 1.7 percent to the overall run time.

The optimizations do become important, however, when grain size is reduced. I wrote a simple Delirium program to test their effectiveness; it is based on a set of mutually recursive functions organized into a loop. Each function iterates several dozen times and calls the next in line, using references to constants bound in an enclosing lexical context. Computation continues for ten iterations through the loop of functions. I compiled the program with and without optimization; the difference in run time was 2.3 milliseconds versus 10.1. For application that are extremely fine-grained with a complex control structure, the optimizations yield an important improvement in execution speed. We are in the process of implementing some search algorithms that have those characteristics.

8 Conclusions

I decided to implement Pythia in Delirium for two main reasons: to test the resilience of our mixed programming paradigm, and to investigate optimization of dataflow programs.

We have proposed a new way to write programs, based on a mixture of functional and imperative programming. The small programs that we wrote to test our original implementation taught us valuable lessons, but ultimately they gave only a superficial measure of the model's usefulness. We have begun implementing realistic applications in this model and in others, comparing the programming difficulties and resulting performance. We are encouraged by the ease with which we converted a motion detection code [6] written in Fortran to run under our system with near linear speed-up.

The compiler is currently the largest Delirium application; it clearly demonstrates that the control structure of a large and complex parallel program can often be expressed compactly in a separate coordination language. The compiler is roughly 5500 lines of code. Of this, 5000 is the same in both the sequential and parallel versions. To switch to the parallel version, we remove a 100 line main module and replace it with the 100 lines of Delirium shown in appendix III and a 400 line auxiliary module that defines the operators. Most of the operator code consists of parallel tree-walking primitives.

Pythia also shows that parallel tree walking is a viable approach to implementing a variety of tree-based computations in parallel. Once I settled on the three tree walking primitives, I quickly grew accustomed to the restrictions they imposed. Because each tree manipulation primitive was only concerned with a single node, the code is fairly short

and was quick to debug. I did all my programming on a single-processor workstation and, once the tree walking primitives had been debugged, the successive versions of the compiler ran immediately in parallel when moved to the Sequent. The deterministic behavior was extremly helpful and I never faced a race condition. Having battled non-determinism many times in past parallel programs, its absence was a welcome change.

While compilation is not representative of the problem domains we are most interested in, Pythia is still a useful application because it involves complex manipulations of a large data structure. We are convinced that the great majority of parallel applications involve such computations and that any useful parallel programming environment must be suitable for expressing them. The functional language community has tended to neglect the issue, although some work has been done on arrays, particularly in SISAL [4]. Work on other data structures has lagged behind. Arvind has proposed the use of lazily updated entities called I-structures [2], but they are difficult to use and to implement.

The second goal of the project was to investigate optimization techniques for dataflow programs. A paper showing the optimization of an intermediate graph description language called IF1 [19] demonstrated how a number of traditional imperative language optimizations can be applied to dataflow graphs. There are some techniques that are no longer useful in the absence of side-effects, but any optimization that reduces the amount of computation in a program is useful in both models.

Because of the characteristics of our existing applications, optimization has not had a major effect on their execution time. We expect some of our currently evolving applications, however, to show a real improvement. As the previous section showed, removing excess closure creation and invocation nodes can have a significant impact on the amount of run time overhead.

Pythia has met both of its goals. It is an effective tool that we have used extensively for the development of a variety of applications, including itself. Parallel tree walking has proven to be an effective way to handle the task of compilation. Graph optimization has yielded significant reductions in run time overhead, minimizing the impact of the system on user code.

References

[1] W.B. Ackerman. "Data Flow Languages,". *Computer*, 15(2), February 1982.

[2] Arvind and Kim P. Gostelow. "An Asynchronous Programming Language and Computing Machine,". Technical Report TR114a, Dept. of Information and Computer Science, University of California, Irvine, December 1978.

[3] Arvind and R.E. Thomas. "I-Structures: An Efficient Data Type for Functional Languages,". Technical Report TM-178, MIT Laboratory for Computer Science, September 1980.

[4] David C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, May 1989.

[5] Cray Research, Inc. *Cray-2 Computer System Functional Description*, hr-2000 edition, 1987.

[6] Frank H. Eeckman, Michael E. Colvin, and Timothy S. Axelrod. "A Retina-Like Model for Motion Detection,". In *IJCNN International Conference on Neural Networks*, pages 247–249, Washington, D.C., 1989.

[7] A.D. Falkoff and K.E. Iverson. "The Design of APL,". In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, pages 240–250. Computer Science Press, Inc., 1987.

[8] Gary Fielland. "The Balance Multiprocessor System,". *IEEE Micro*, 1(8):57–69, February 1988.

[9] C. N. Fischer. *On Parsing Context Free Languages in Parallel Environments.* PhD thesis, Cornell University, 1975.

[10] John Gurd, C.C. Kirkham, and Ian Watson. "The Manchester Prototype Dataflow Computer,". *Communications of the ACM*, 28(1):34–52, January 1985.

[11] M.D. Hill, S.J. Eggers, J.R. Larus, G.S. Taylor, G. Adams, B.K. Bose, G.A. Gibson, P.M. Hansen, J. Keller, S.I. Kong, C.G. Lee, D. Lee, J.M. Pendleton, S.A. Ritchie, D.A. Wood, B.G. Zorn, P.N. Hilfinger, D.A. Hodges, R.H. Katz, J.K. Ousterhout, and D.A. Patterson. "Design Decisions in SPUR,". *Computer*, 19(11), November 1986.

[12] Paul Hudak. "ALFL Reference Manual and Programmers Guide, 2nd edition,". Technical Report YALEU/DCS/TR-322, Yale University, October 1984.

[13] Edward A. Lee and D.G. Messerschmitt. "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing,". *IEEE Transactions on on Computers*, C-36(2), January 1982.

[14] Steven Lucco and Oliver Sharp. "Madness: A Parallel Programming Environment,". Technical Report in preparation, UC/Berkeley Computer Science Department, 1990.

[15] James R. McGraw. "The VAL Language: Description and Analysis,". *ACM Transactions on on Programming Languages and Systems*, 4(1):44–82, January 1982.

[16] James R. McGraw and Stephen K. Skedzielewski. "Streams and Iteration in Val: Additions to a Data Flow Language,". In *Proceedings of the Third International Conference on Distributed Computer Systems*, pages 730–739, October 1982.

[17] Jonathan Rees and William Clinger. "Revised^3 Report on the Algorithmic Language SCHEME,". *SIGPLAN Notices*, 21(12), December 1986.

[18] Vivek Sarkar and John Hennessey. "Partitioning Parallel Programs for Macro Dataflow,". In *ACM Conference on Lisp and Functional Programming*, pages 202–211, Cambridge, Mass., 1986.

[19] S.K. Skedzielewski and M.L. Welcome. "Data Flow Graph Optimization in IF1,". In *Functional Programming Languages and Computer Architecture*, pages 17–34, Nancy, France, 1985.

[20] Charles P. Thacker, Lawrence C. Stewart, and Jr. Edwin H. Satterthwaite. "Firefly: A Multiprocessor Workstation,". Technical Report 23, DEC SRC, December 1987.

[21] Chun Pong Yu. "Practical Parallel Lexing,". Master's thesis, Computer Systems Research Institute, University of Toronto, May 1989.

[22] M. Zosel. "A Parallel Approach to Compilation,". In *ACM Symposium on Principles of Programming Languages*, pages 59–70, 1973.

Figure 1:
Creation of a Closure

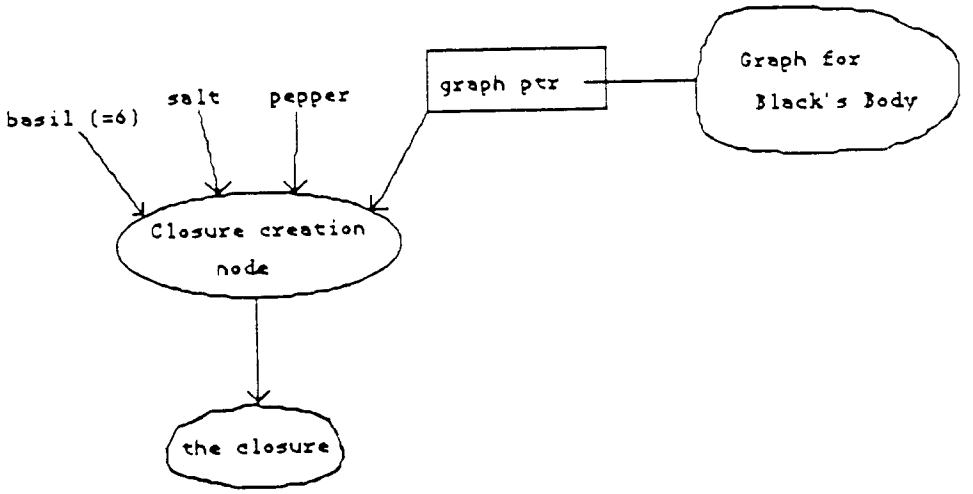


Figure 2:
The Expander in Action

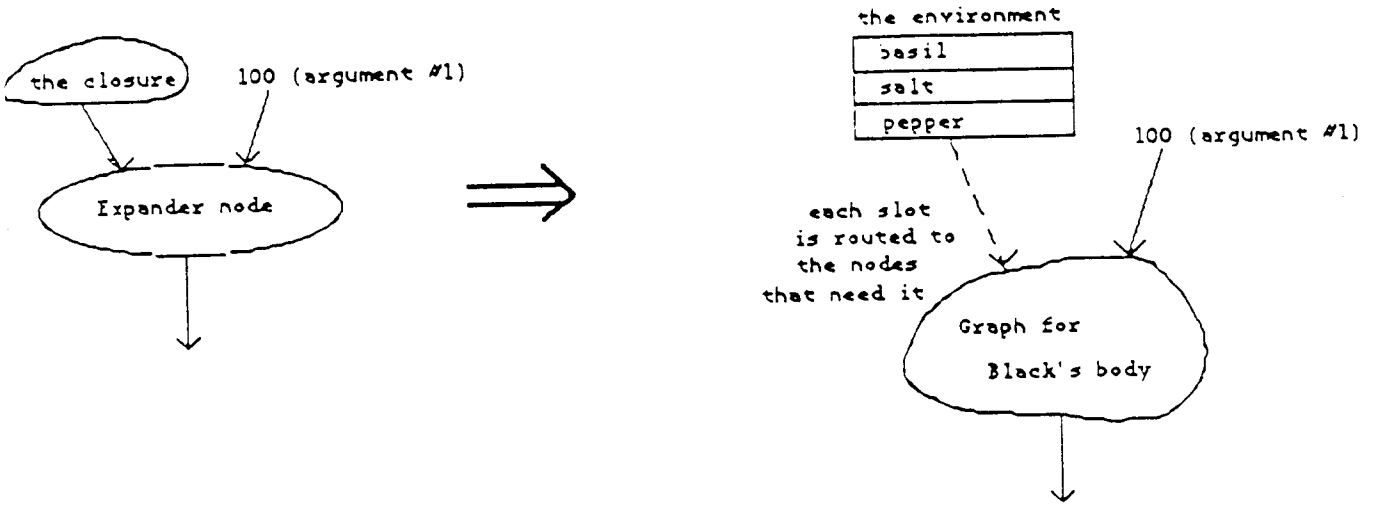


Figure 3:
Compilation of a Let Statement

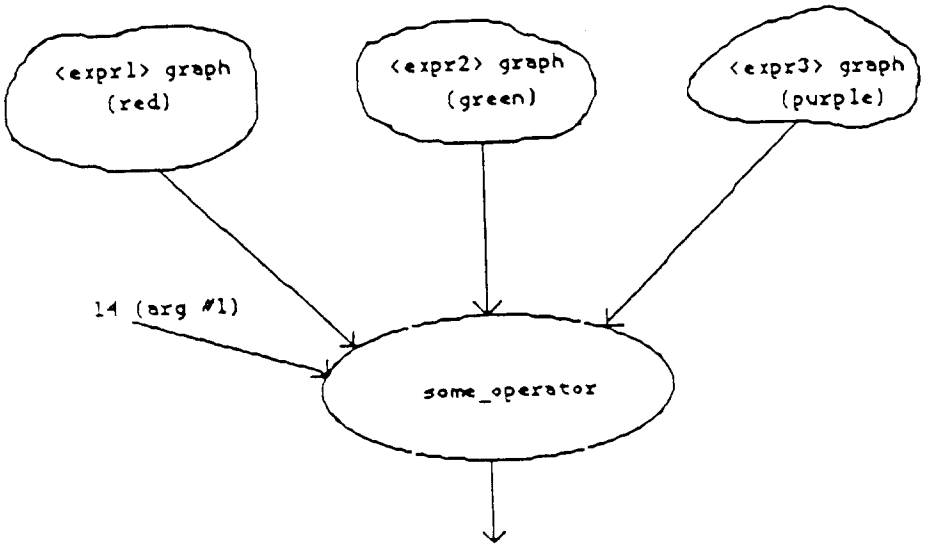
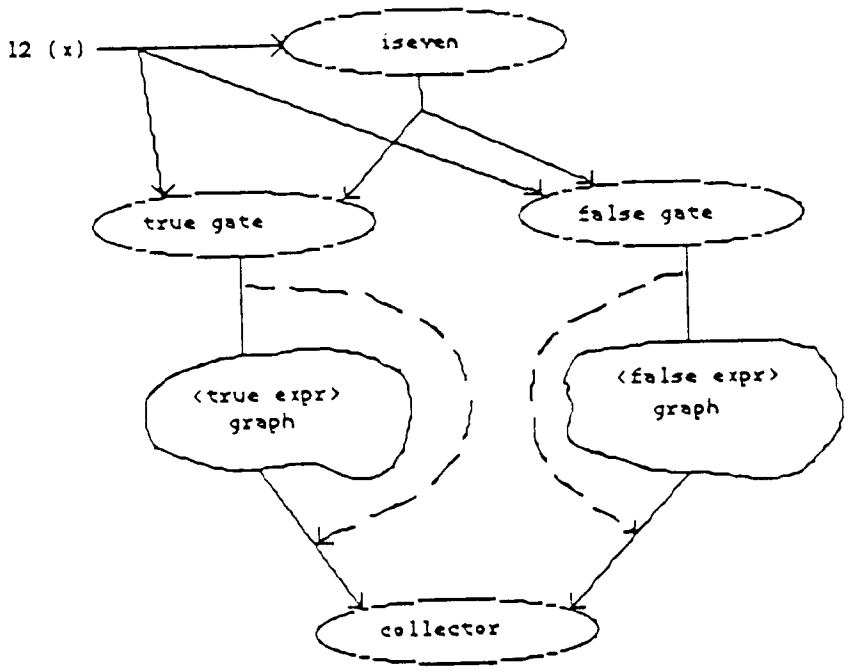


Figure 4: Conditional Statment



Note: the dashed arrows indicate null token forwarding

10 Appendix II: BNF Grammar

```
program : function_list

function_list : function | macro | function_list function
              | function_list macro

macro : 'macro' id args expr

function : id args expr

args : '(' ')'  |  '(' arg_list ')'

arg_list : arg_list ',' id  |  id

expr : conditional | let_stmt | iterate | mult_value | func_app
       | macro_call | prim_expr

let_stmt : LET bindings IN expr

bindings : bindings binding | binding

binding : var_binding | mult_var_binding  | func_def

var_binding : id '=' expr

mult_var_binding : '<' var_list '>' '=' expr

func_def : id args '=' expr

var_list : var_list ',' id | id

iterate : ITERATE '[' iter_bindings ']' WHILE expr ',' RESULT expr

iter_bindings : iter_bindings iter_binding | iter_binding

iter_binding : id '=' expr ',' expr

conditional : IF expr THEN expr ELSE expr

prim_expr : integer | string | id | float

mult_value : '<' expr_list '>'
```

```
expr_list : expr_list ',' expr | expr

func_app : id app_args

macro_call : id app_args

app_args : '(' ')' | '(' app_arg_list ')'

app_arg_list : app_arg_list ',' expr | expr
```

11 Appendix III: Pythia – the Delirium Code

```
#  pythia.del - this is the Delirium code for the compiler

macro tree_walk_synthesize(tree,operator,extra)
  let <st1,st2,st3> = tree_syn_chop(tree)
  in tree_syn_merge(tree_syn_op(st1,operator,extra),
                    tree_syn_op(st2,operator,extra),
                    tree_syn_op(st3,operator,extra),
              operator,extra)

macro tree_walk_update(tree,node_type,operator,extra)
  let <ut1,ut2,ut3> = tree_up_chop(tree,node_type,operator,extra)
  in tree_merge(tree_up_op(ut1,node_type,operator,extra),
                tree_up_op(ut2,node_type,operator,extra),
                tree_up_op(ut3,node_type,operator,extra))

macro tree_walk_inherit(tree,operator,info,extra)
  let <it1,it2,it3> = tree_in_chop(tree,operator,info,extra)
  in tree_merge(tree_in_op(it1,operator,extra),
                tree_in_op(it2,operator,extra),
                tree_in_op(it3,operator,extra))

#  iterate_to_fixpoint is a macro that iterates synthesized attribute
# tree traversals until the second return value is false

macro iterate_to_fixpoint(tree,operator,extra)
  iterate {
    results = <tree,TRUE>,
              tree_walk_synthesize(car(results),operator,extra)
  } while cadr(results),
    result car(results)

main()
  let <init_tree,macros> = compile()
      <ops,types> = read_operator_info()
      expand_tree = iterate_to_fixpoint(init_tree,expand_macro,macros)
      analyzed_tree = analyze(expand_tree,ops,macros)
      optimized_tree = optimize(analyzed_tree,ops)
      used_tree = mark_used(optimized_tree)
      <convert_tree,number> = convert(used_tree)
      cleaned_tree = tree_walk_update(convert_tree,LETREC,clean_up,NULL)
  in output_graph(cleaned_tree,number,types,ops)
```

```
#  compile - read in the lexemes, do a parallel YACC parse, recombine
# the parse tree, and return it.  Return the merged macro table as well.

compile()
  let <c1,c2,c3> = split_lexemes()
  in forge_parse_tree(partial_parse(c1),partial_parse(c2),
                      partial_parse(c3))

#  analyze - given a parse tree, the operator descriptions, and the
# hash table of macros, do environment analysis and return the
# annotated tree.

analyze(tree,ops,macros)
  let free_tree = car(tree_walk_synthesize(tree,gather_free,NULL))
      recon_tree = tree_walk_inherit(free_tree,reconcile_refs,ops,NULL)
      marked_tree = tree_walk_update(recon_tree,ANY,mark_extra,SAFE)
      pruned_tree = tree_walk_update(marked_tree,ANY,prune_extra,SAFE)
      overmark_tree = tree_walk_update(pruned_tree,ANY,mark_extra,RISK)
      closed_tree = iterate_to_fixpoint(overmark_tree,mark_closures,NULL)
      final_tree = tree_walk_update(closed_tree,ANY,mark_extra,FINISH)
  in tree_walk_update(final_tree,ANY,prune_extra,FINISH)

#  optimize - first iterate the inline expansion code until fixpoint.
# Then do constant propagation and common sub-expression elimination.

optimize(tree,ops)
  let expanded_tree =
        iterate {
          results = do_an_expansion(tree,ops),
                    do_an_expansion(car(results),ops)
        } while cadr(results),
          result car(results)
      const_tree = iterate_to_fixpoint(expanded_tree,propagate_const,NULL)
  in tree_walk_update(const_tree,ANY,CSE,NULL)

# do_an_expansion - given a tree and the operators, do an inline
#   expansion pass.  Then mark any further expansions that are to be
#   done.  Return the newly expanded tree and a TRUE/FALSE boolean
#   indicating whether more work remains.

do_an_expansion(tree,ops)
  let modified_tree = tree_walk_update(tree,FUNC_APP,expand_calls,ops)
      redone_tree = analyze(modified_tree,ops,NULL)
  in tree_walk_synthesize(redone_tree,mark_2_expand,NULL)
```

```
#  convert - given a parse tree, number each template and then convert
# it into a graph.  Return the annotated tree and the total number of
# templates.

convert(tree)
  let <number,numbered_tree> = number_templates(tree,0)
      <ct1,ct2,ct3> = output_chop(numbered_tree)
      new_tree = tree_merge(output_op(ct1), output_op(ct2),
                            output_op(ct3))
  in <new_tree,number>
```

12 Appendix IV: BNF For Intermediate Form

```
graph : 'ntemplates:' int 'start_template:' int templates

templates : 'template index:' stats arg_arcs env_arcs const_arcs nodes

stats : 'n_env:' int 'n_const:' int 'n_args:' int 'n_arcs:' int

arg_arcs : arc-set

env_arcs : arc-set

const_arcs : const_arcs const_arc | null

const_arc : arc 'constant type: ' int ' data: ' const_data

arc-set : 'ncopies:' int arcs

arcs : arc arcs | null

arc : 'arc num' int 'to' int 'arc is word:' flag 'arc is destr:' flag
      'fwd null: ' int int

node : 'node' int 'intr: ' int 'type:' int 'template: ' int 'name:' id
       'noutputs:' n outputs

outputs : arc-set outputs | null

flag : 1 | 0
```