A PRAGMATIC APPROACH TO THE ANALYSIS AND
COMPILED OF LAZY FUNCTIONAL LANGUAGES

Hugh Glaser, Pieter Hartel*, John Wild

Department of Electronics and Computer Science
University of Southampton,
Southampton,
UK

The aim of the FAST Project is to provide an implementation of a functional
language, Haskell, on a transputer array. An important component of the system
is a highly optimising compiler for Haskell to a single transputer. This paper
presents a methodology for exploitation of many standard and some new tech-
niques in a clear and concise notation. Results are included showing that the
optimisations give significant improvement over the standard combinator and
(Johnsson’s 1984) G-machine implementations.

Introduction

The FAST (Functional programming for ArrayS of Transputers) Project, funded by the UK gov-
ernment, is a collaboration between the University of Southampton, Imperial College, London
and Meiko Ltd. of Bristol. The aim is to provide an implementation of a pure, lazy, functional
language such as Haskell [12] on transputer arrays. The methodology for distribution is a variant
of the process description language, Caliban [15], and requires a highly-optimised implemen-
tation of Haskell on a single transputer as one of the components. The work reported here is on
the development of the pilot compiler, that generates C.

The rest of the paper is organised as follows. In the remainder of the introduction we explain
the terminology and concepts required. The next section gives a description of the flow graphs
used in the compiler, and this is followed by an example of the analysis of the graphs, and some
details of the implementation of the analytical schemes. We then show how this analysis can be
used to synthesise code for a conventional machine, using C as the target, and finally we present
preliminary results and conclusions.

*Pieter Hartel is on sabbatical leave from the University of Amsterdam
1.1 Functional Languages

Pure, lazy functional languages such as Miranda [20] and Haskell [12] are claimed to have a number of advantages over procedural languages of the Algol and Fortran type. In particular they:

1. have a good formal basis, (the λ-calculus);
2. are higher level, relieving the programmer of some of the detail of programming;
3. have no side effects during execution, facilitating parallel evaluation of programs.

As an example, consider the function append, which joins two lists, written in the functional language Haskell shown in figure 1.

```
append :: [a] -> [a] -> [a]
append []    second = second
append (head:tail) second = head : (append tail second)
```

Figure 1: append function

The first line of this function is the type specification. It defines append to take two arguments, both lists of (the same) unspecified type, and return a list of the same type. The second line says that the append of an empty list to any list is simply the second list. The third line uses the infix operator ";", both on the left hand side to pattern match on the argument, and also on the right hand side as a constructor.

We can see here that:

1. append is a pure function (in the sense that it has no side effects), facilitating the use of formal manipulation tools;
2. the order of evaluation is not explicitly stated, as expressions evaluate only when required, and then only once (lazy evaluation [11]); any allocation and reclamation of storage is left to the system (garbage collection [5]);
3. the interface to the outside world is completely specified by the parameters and result.

Since the second of these points implies that the programmer is relieved of the burden of much of the book-keeping associated with the actual computation, and the third implies that functions such as append can be executed on a processor with little concern for values being changed by other processors, it can be seen that functional languages have a number of advantages [13]. Unfortunately, there is a price to be paid for these advantages.

In relieving the programmer of the burden of specifying evaluation order and storage allocation, the system requires greater intelligence on the part of the implementation. In practice, this means that the implementation should support a suitable execution model for lazy evaluation, such as graph reduction, and have an efficient storage management system.
Graph Reduction

Graph reduction [23] is an abstract machine for evaluating functional languages, maintaining semantics of lazy evaluation, and forms the basis for many implementations of functional languages [1, 6, 7, 18]. As a small example, consider the evaluation of `append` with the arguments `[1]` and `[2]` (where `[1]` represents the list with the single element, 1, as shown in re 2.)

![Figure 2: Reduction of append [1] [2] to yield [1,2].](image)

We see here how the graph that represents the application of `append` to its arguments is reduced in two stages, using the rules of the program, to the graph that represents the required result. (The application nodes that are selected for reduction and subsequently updated have a box around them for identification.) There are a number of issues involved in correctly implementing graph reduction. For example the second argument to `append` is never examined, should therefore not be evaluated if it is as yet unevaluated, and there is a need to update a node of the graph as it is evaluated, in case it is shared with another expression. A thorough treatment of many of the issues can be found in [17]. Although providing an excellent basis for evaluating functional programs, pure graph reduction incurs a number of performance penalties:

- Evaluation switches between contexts, as different parts of the graph are required;
- Delayed computations, such as the second argument to `append`, can make greater demands on storage and processor power than early reduction;
- Updating unshared nodes is wasteful, but not always avoidable;
- Examining nodes to find out whether they have already been reduced is also an overhead;
- Since the underlying target machines do not have a "graph reducing" instruction set, there is an element of interpretation of the graph.

One of the ways of avoiding these penalties is to avoid interpreting the graph whenever possible. This reduces the context switching, as well as making the underlying evaluation mechanism more efficient. It is possible to do this because the graph reduction machine has no separate cycle when calling a function: Firstly, the graph corresponding to the function body (left-hand side) is created, and secondly this graph is reduced. By effectively performing graph creation at compile time, it is sometimes possible to compile a graph into native machine code, short-circuiting the graph reducer altogether. This adds the bonus of reducing run time graph creation, which helps to avoid heap cell
Heap cell claims are expensive [8], so such optimisations are important. To perform this optimisation, it is necessary to analyse the program to determine when an expression can be evaluated at graph construction time instead of building a suspension which will be reduced later. Analyses of this type (strictness analysis) are a major topic of research in functional language implementation. We perform static analysis in much the same way as Wray[24], which approximates to the information gained by abstract interpretive methods.

Another area in which improvements can be made is to avoid examining nodes to see if they have been evaluated. The overhead of this operation is twofold. Firstly it requires work to examine the node, and secondly it requires "tags", or some other method, so that evaluated and unevaluated nodes can be distinguished. Various methods have been proposed to reduce this penalty [17], and our system makes strenuous attempts to keep it to a minimum.

1.3 The FAST Compiler

Analysis is an important part of the optimisation process. The other part of the process is the synthesis, based on the analysis, of efficient, correct code. This is an important, separate part of the compilation. On one hand it is not always the case that the code generation strategy is able to take full advantage of a sophisticated analysis, and on the other hand, it might be that, although the analysis permits a particular reduction strategy, the synthesis decides a variant is the most efficient.

From the outset, it was decided that the analysis and synthesis should be specified within a single formal framework. This allows experimentation with different styles of analysis and synthesis. We have been able to provide a clear description of a number of analyses and syntheses, such as static analysis for strictness [24]; compiling code to perform reductions before they are needed, passing arguments on the stack instead of heap cell pointers and cheap weak head normal form analysis [14]; value representation [17].

Before showing an example of an analysis scheme (that for static strictness analysis), we first show the flow graph that is used to represent the program at compile time. It is worth commenting that this graph is not the same as the graph that will be (abstractly) reduced. The reader should rather view this graph as a recipe for creating the reducible graph at run time.

2 Flow Graphs

A flow graph represents a functional program that is being compiled. The representation we use is slightly richer than a standard syntax tree. Since much of the program analysis and code synthesis is based on value representations, it is convenient keep explicit track of the generators and consumers of each value (cf. defuse information in conventional compilers). We therefore provide SWITCH and USE nodes to explicitly route parameters and other data.

Our flow graphs contain 11 kinds of nodes; each corresponds to a primitive operation. The edges of a flow graph are numeric labels. The input language to the compiler is somewhat richer than the lambda-calculus. The two standard axioms of the lambda-calculus, application and abstraction are present in the flow graph as BIND and LAMBDA nodes. There is a facility for creating primitive data objects, the SOURCE. There is also a primitive SINK for destroying an object. SOURCE and SINK are corresponding nodes, as are LAMBDA and BIND. Instead of allowing indiscriminate copying of objects, whether applications, abstractions or primitive data objects we require USE nodes to specify that a particular object is used at more than one place.
ly speaking this is all that is required for a flow graph representation of the untyped $\lambda$-

lus. However a small number of extra primitives are highly desirable. These are SWITCH

\texttt{MERGE} to represent IF ... THEN ... ELSE ... \texttt{FI}; IMPORT and EXPORT to regulate

exchange of information across function boundaries; CALL and RESULT to present the

user with information about the presence of fully parameterised calls, as opposed to partial

ations, and fully built functions. The example of figure 3 shows the flow graph for the

itrous factorial function.

Figure 3: Flow graph of $\text{fac} = \lambda n. IF \ n \leq 0 \ THEN \ 1 \ ELSE \ n \times \text{fac} (n-1) \ FI$
The flow graph of figure 3 may be given a data driven (call-by-value) reading as well as a demand driven (call-by-need) reading. We found this dichotomy convenient to reason about programs. The difference between our interpretation and that used in general data flow work [21] is that we do not assume that a node need actually compute a primitive result in order to fire. Nodes are also able to form suspensions (closures) of the operator and operands. To make the flow graph as data-driven as possible we need compile time analysis to work out when to fire computations.

First we show that by giving the flow graph a demand driven reading we are able to derive an early point at which it is safe to evaluate an expression. The starting point is the function output on edge 34 of the function. The demand flow is complementary to the data flow and it travels in the opposite direction on the edges.

The fundamental observation is that, in a lazy graph reduction system, an expression is never evaluated unless its value (as opposed to a suspension) is actually required. This means that whenever an application of factorial is evaluated, its value is definitely required. In the example of demand flow reading that we will now present, it is this static strict property that travels upwards through the graph.

The first node to be considered is the MERGE node. As we require a value on its output (edge 34), we must also have values on its three inputs (edges 31, 32 and 33). The USE nodes merely pass information on, such that we know that a value is required on edge 11. Now examining basic block 1 we see that it imports a function, "≤", that is bound to two arguments. The first BIND creates a partial application on edge 9. This is subsequently bound to the constant on edge 8, where it becomes a fully parameterised call on edge 10. This call can in principle be evaluated.

In the current example it will indeed be evaluated immediately, because we have just derived that a value is required on edge 11. In a non-strict context the strand of BIND nodes terminated by a CALL must arrange for a suspension to be created in the heap, for possible evaluation later. The imported function "≤" is known to be strict in its two inputs. This implies that to produce an output value, the function "≤" must be presented with values on its inputs. The requirement for a value on edge 8 is immediately satisfied because a SOURCE always generates a value. The other input to the function "≤" receives its value via a USE as the argument to factorial.

In conclusion we must have at least a value as input for factorial to produce a value as output. Factorial is therefore strict in its input.

The strict requirement propagated by the MERGE enters the other two basic blocks at edges 31 and 32. At first sight this appears strange because it looks like we are evaluating both the then- and else-branch of the conditional. This is not the case however, because the condition will be evaluated before the then- or else-branch. The truth value determines which branch will be entered and the other will be ignored. The compile time analysis may thus safely assume that both branches produce a value.

The SOURCE always produces a value on edge 32 and on edge 31 we require the result of a multiplication. Because the function '×' is strict in its two arguments (known), we must have values on edges 16 and 28. This presents us with a slight problem in that we are currently analysing the function for which a recursive call is encountered. Fortunately we have just discovered that factorial is strict in its argument, based on the requirements of the MERGE and basic block 1. Therefore the strict property propagates to edge 25, and also to edges 22 and 19, since '-' is also fully strict. Note that we could have analysed basic block 2 before basic block 1. This would have prevented us from determining that factorial is strict before we require the information. In this case we would have had to build a suspension for the call to '-' and leave it to factorial to force evaluation of its argument when entered. The termination properties
A More Formal Treatment of Flow Graphs

A demand driven reading that works out how to order computations is now described in a more formal way. A module contains a set of related function definitions. It is treated as a unit in compilation. Each function is represented as a sub-graph. The functions are connected via USE and USE nodes. All edges in a compilation unit are uniquely numbered. The nodes are represented as 4-tuples of the form \((\text{inputs, type, qualifiers, outputs})\). Here, inputs and outputs are sets of edges, types indicate which of the 11 different node types the 4-tuple represents. Each edge carries a list of further qualifiers that represent synthesized information pertinent to the edge. Each phase of the analysis adds further elements to the set of names and modifies or adds existing elements as appropriate. The elements particular to a phase of the analysis are called attributes. This mechanism is loosely based on the \(\Gamma\) mode of computation [2]. An attribute is a triple of the form \((\text{type, edge, value})\). The type specifies which attribute the value represents on the particular edge. For instance \((\text{STR, E, T})\) means that edge 6 carries the strict value \(T=\text{TRUE}\). Transformations on the multiset nodes and attributes are achieved by means of rewrite rules with pattern matching of multiset elements or components \(\text{reset}\). The set of rules that deal with one particular attribute is called a compilation unit.

Consider the compilation scheme for the strict property as shown in Figure 4.

The clauses of the function \(\text{STR}\) when applied to a multiset are tried top down, left to right elements of the multiset. The words spelled in upper case are literals that must be present shown, lower case words represent variables that can be bound to values by the pattern matching. Repeated variables of the same name must all match the same value. Semicolons separate elements of (multi)sets while commas separate the elements of lists. Multisets as a whole are enclosed in curly brackets and lists in round brackets. For example, if the three multiset elements as specified in clause 1 (i.e. the node \((a, \text{LAMBDA}, p, (l, c))\) and the attributes \(R, \text{l}, \text{s1}\) and \((\text{STR, c, c})\) are present, a match occurs of that clause. The following bindings will then be made for the duration of the current invocation of the function \(\text{STR}: a\) represents the head edge of the LAMBDA node and \(l\) and \(c\) its output edges. \(p\) is the qualifier that applies
1. Str \{a, LAMBDA, p, (i, 0); STR, l, t, STR, a, sc, x\} = Str \{STR, a, sc, x\} \cup +\{x\}
2. Str \{(a, (b), BIND, p, c); ABS, a, (n, \{n + 1, t, 1\}); STR, l, t, STR, c, t, x\} = Str \{STR, a, t, STR, b, t, STR, l, t, x\} \cup +\{-\{STR, l, t\}\}
3. Str \{(a, b), BIND, p, c; STR, c, sc, x\} = Str \{STR, a, sc, STR, b, p, x\} \cup +\{-x\}
4. Str \{(a, b), SWITCH, p, c, d; STR, c, sc, STR, d, sd, x\} = Str \{STR, a, p, STR, b, sc & sd, x\} \cup +\{-x\}
5. Str \{(a, b, c), MERGE, p, d; STR, d, sd, x\} = Str \{STR, a, sd, STR, b, sd, STR, c, sd, x\} \cup +\{-x\}
6. Str \{SOURCE, p, a; STR, a, sc, x\} = Str \{x\} \cup +\{-x\}
7. Str \{a, SINK, p, ; x\} = Str \{STR, a, p, x\} \cup +\{-x\}
8. Str \{IMPORT, p, a; STR, a, sc, x\} = Str \{x\} \cup +\{-x\}
9. Str \{a, EXPORT, p, ; x\} = Str \{STR, a, p, x\} \cup +\{-x\}
10. Str \{a, CALL, p, b; STR, b, sb, x\} = Str \{STR, a, bk, x\} \cup +\{-x\}
11. Str \{a, RESULT, p, b; x\} = Str \{STR, a, t, x\} \cup +\{-x\}
12. Str \{a, USE, p_1, b_1, ..., a, USE, p_n, bn; STR, b_1, ..., STR, b_n, sbn, x\} = Str \{STR, a, sb_1, ..., sb_n, x\} \cup +\{-x\}
13. Str x = x

Figure 4: Compilation scheme for the "strict" property
A LAMBDA node; \( sl \) and \( sc \) are the strictness values on edges \( l \) and \( c \) respectively. The ble \( x \) is bound to the the remaining elements of the multiset. We found it convenient to also symbol 'w' that represents the multiset as a whole as a default binding. It could be made city, in the case of clause 1 for example as: WHERE \( w = \{a,LAMBDA,p,\{l,c\}; STR,l,sl; c,sc,tx\} \). The rewrite that takes place after the match of clause 1 has succeeded, generates a attribute \( (STR,a,sc) \) for the input edge of the LAMBDA node. Its strictness value is merely a from the o-output of the LAMBDA node. After the rewrite, further matches are attempted the union of the newly generated element \( \{STR,a,sc\} \) and the multiset \( \{x\} \). The remaining \( \text{ents} \leftarrow \{x\} \) are of no further concern to the Str scheme.

Clause 2 of the Str compilation scheme shows how we can work out whether the current argument bound by the BIND node is a strict argument. This uses an attribute called (for abstraction). It records for each actual argument \( b \) input edge to a BIND node to which aal argument \( f \) output edge of a LAMBDA node it corresponds. The reason that there are clauses present for the BIND node is that, depending on the order in which elements of the set that represents a module are processed by the Str function, the strictness of a formal element may or may not yet be known before the information is required. The first BIND se (line 2) applies when the information is already known, the second (line 3) otherwise.

The USE clause at line 12 requires some further explanation. Here we have used ellipses match all nodes from the multiset that specify the USE of a particular edge \( a \). The qualifiers output edges are bound to the variables \( pl \ldots pn \) and \( bl \ldots bn \) respectively. The variable \( n \) xed to the number of USE nodes found, where \( n \) is a positive integer. The attribute for the \( \{a\} \) is the logical conjunction of \( sb1 \ldots sbn \) of the attributes on the output edges of all the \( 1 \) nodes.

Rewriting goes on until none of the clauses 1 — 12 apply. The clause at line 13 then returns \( t \) is left of the original multiset and the Str scheme has done its work.

The quality of the strictness analysis depends on the order in which multiset elements are ed for matching. This has the advantage that the method is cheap to implement; its plexity is of \( O(n) \) where \( n \) is the number of nodes in the flow graph. Since the strictness mation is only an approximation, we can improve the information by re-applying the ion Str to its own output, perhaps repeating this process several times. For the programs we have analysed so far we found no further improvements after 4 runs of the analysis.

etically, plateaus present severe obstacles to the derivation of complete first order strictness at domains [4]. We are not particularly worried by this phenomenon, as the analysis so far us to experiment effectively with the problems of synthesis, and the system allows the ot of strictness information from more powerful analysers. Pattern matching often causes ations to make tests on the form of their arguments, such that our simple strictness analyser covers these arguments as strict. There are examples such as the one given by Clack et al [4].

Our analysis can not approximate well: \( fx = IF (x = 0) \) THEN \( y \) ELSE \( (f(x-1)) \) Fl. Even th is function is strict in both arguments our analysis will only discover it to be strict in \( x \).

The Implementation of a Compilation Scheme

A description of the schemes is in a convenient form for understanding, but pattern matching arge multisets is not a sensible implementation method. We manually translate the schemes a programming language, and synthesise the attributes in a more algorithmic fashion.

Firstly we number edges such that, with one exception, the input edges to a node bear lower abers than the output edges. The exception is the RESULT node, which is treated in a special
way. We then order the nodes in our multiset according to their minimum input edge. The nodes
that represent the factorial function are shown ordered in figure 5. The implementation of the
Str function then takes the node with the largest input edge, which in the case of the factorial
element is the RESULT node. It does not need the presence of any attributes so the multiset
may be rewritten by the clause at line 11 of the compilation scheme from the state shown in the
left column of figure 5 to that shown on the right.

1. LAMBDA(), (2,3)  1. LAMBDA(), (2,3)
2. USE(),  5  2. USE(),  5
3. USE(),  12  3. USE(),  12
3. IMPORT, 0  3. IMPORT, 0
0. BIND, 9  0. BIND, 9
0. SOURCE, 8  0. SOURCE, 8
(6,5,9,8)  (6,5,9,8)
(9,8)  (9,8)
10. CALL, 11  10. CALL, 11
11. USE, 13  11. USE, 13
11. USE, 33  11. USE, 33
(13,12,13,12) (14,15,14,15)
14. SINK, 0  14. SINK, 0
15. USE, 16  15. USE, 16
15. USE, 19  15. USE, 19
0. IMPORT, 17  0. IMPORT, 17
(17,16)  (17,16)
0. BIND, 29  0. BIND, 29
0. IMPORT, 20  0. IMPORT, 20
(20,19)  (20,19)
0. BIND, 23  0. BIND, 23
0. SOURCE, I, 22  0. SOURCE, I, 22
(23,22,23,22) (23,22,23,23)
24. CALL, 24  24. CALL, 24
0. IMPORT, 25  0. IMPORT, 25
(25,26)  (25,26)
27. CALL, 27  27. CALL, 27
(29,28,29,28)  (29,28,29,29)
30. CALL, 30  30. CALL, 30
0. SOURCE, I, 32  0. SOURCE, I, 32
(33,32,31,33,32,31,32) (33,32,31,33,32,31,32)
34. RESULT, 1  34. STR, 34, T

Figure 5: First two stages in the Str analysis of the factorial function

The first iteration of the Str rewriting terminates when the STR attribute has been calculated
for all the edges. Factorial requires a second iteration to yield optimal strictness information.

Our prototype compiler is written in SASL [19]. Compilation schemes are implemented
as recursive functions over a list of nodes and attributes. The ordering of the nodes makes it
possible to use ordinary pattern matching on lists instead of multisets. The number of nodes is
hardly affected by the compilation schemes (constant folding may remove some nodes.) It is
thus possible to represent the multiset as an array rather than a list. (Our local variant of SASL
provides support for some array operations.) We found that this increases compilation speed by
an order of magnitude.
5 Run Time Organisation

The prototype compiler transforms a functional program into an equivalent program in an imperative language. We use C as our output at present, but any other language that supports recursion would have been adequate. This use of an efficient high level language gives a more effective development environment than generating machine code directly. It allows good instrumentation and experimentation on significant source programs. The imperative program contains calls to runtime routines that implement standard graph reduction when required. Some user functions can be compiled into a form that does not require graph reduction at all. The function factorial for instance is known to be strict, such that when passing it an argument the compiler can arrange the code such that the argument is evaluated before factorial is called. To take full advantage of call by value parameter passing we decided to support several parameter passing mechanisms. The guiding principle is that the callee decides in what form it wants the arguments. The function \texttt{fac} of figure 6 shows the C code produced for the flow graph of figure 3. Of particular interest is the form taken by the recursive call: \texttt{fac (a - 1)}. The imperative compiler generates code to subtract one from \texttt{a} and passes the difference to the recursive invocation of \texttt{fac}. No heap cells are allocated or even accessed throughout the execution of a call to \texttt{fac}.

```c
int fac (a)
int x;
{
    int t1;
    t1 = a <= 1;
    if(t1) {
        return 1;
    } else {
        return a * fac(a - 1);
    }
}
```

Figure 6: Compiler output for factorial

For calls to factorial that appear in a non-strict context the (functional) compiler must generate code that builds a suspension. This is a canned form of a call to a function with (some of) its arguments supplied. The runtime support contains functions to build such suspensions and an \texttt{unwind} function to evaluate a suspension. Letting the the caller decide on the form of the arguments poses a slight problem here. The C function of figure 6 expects its argument as a value on the stack. The standard unwind as it is usually implemented in graph reduction systems provides the function it fires up with access to its arguments via pointers from the stack to the heap. We have solved this incompatibility by generating several versions of each user function, in principle one for each different usage. There are several compilation schemes in the compiler that allow it to work out which version of a function fits a particular usage. When functions are manipulated by higher order functions, the compiler does not have information about the required form of the parameters to a call, so it has to assume that all arguments are in the heap and the compiler uses the version of the function that expects its arguments in the heap. There is never any runtime interpretation required to match the right type of function to a particular
call. It was found that for many function calls straight function calls can be generated, such that the default suspend/unwind mechanism is not used as often as in pure graph reduction systems. The problem with this approach is that many different versions of the same function may be required. In practice, however, we found that most functions have only a few call sites; for the 1500 functions in the benchmark programs the average number of different ways a function is used is 1.54. Multiple copies of functions can be simulated by the use of multiple entry points.

6 Preliminary Results

We have performed some experiments with our prototype compiler to assess the performance of our method. The programs that we have available are the result of collecting performance data on implementations of functional programming languages [9, 10]. The programs we have used so far are:

1. fib 7 prints the seventh Fibonacci number using double recursion;
2. hamming 100 prints, in ascending order, the first 100 natural numbers whose prime factors are 2, 3 and 5;
3. em script runs a simple script through a functional implementation of the UNIX text editor; [16]
4. gc ode compiles the qsort program into scalar G-machine code according to the compilation schemes as described in Johnsson’s paper [14];
5. lambda ( S K K ) evaluates to I on an implementation of the λ-K calculus; [9]
6. qsort (sin 1, ..., sin 1024) sorts a list of 1024 numbers using quick sort;
7. sched 7 calculates an optimum schedule of 7 parallel jobs with a branch and bound algorithm; [22]
8. wave 3 predicts the tides in a rectangular estuary of the North Sea over a period of 3x20 minutes. [22]

The statistic that provides the best indication for the quality of our method is the number of cell claims made. Table 1 shows the cell claims (as reported in [9, 10]) witnessed by Turner's standard combinator reduction machine [18], a scalar version of Johnsson’s G-machine [14] and the C code produced by the flow graph compiler. The numbers apply to evaluation only. The cell claims required to build the initial expressions are not included. Our implementation method does not increase the cost of ordinary lazy graph reduction, such that we may safely assume that when the number of cell claims has been reduced, the total execution time will also be lower. With the exception of wave, the number of primitive operations (multiplications, divisions etc) performed by a benchmark is similar on all three implementation methods.

The columns in table 1 show a decrease in the number of claimed cells when moving from standard combinators via the G-machine to the flow graph compiler. The most important difference between these implementations is the use of strictness information. Standard combinators do not use strictness information at all. The other methods use strictness information about weak head normal forms. The G-machine only uses it locally within a function, while the flow
graph compiler carries weak head strictness across function boundaries. The *hamming* program, which is notorious for its use of lazy lists, does not benefit much from extending the analysis across function boundaries. The other programs exhibit varying degrees of success for extending analysis across function boundaries.

<table>
<thead>
<tr>
<th></th>
<th>fib</th>
<th>hamming</th>
<th>em</th>
<th>gcode</th>
<th>lambda</th>
<th>qsort</th>
<th>sched</th>
<th>wave</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td>289</td>
<td>15970</td>
<td>733742</td>
<td>966963</td>
<td>71128</td>
<td>1131047</td>
<td>477243</td>
<td>424623</td>
</tr>
</tbody>
</table>

Table 1: Cell claims for example programs

Table 2: Reduction in cell claims for example programs

<table>
<thead>
<tr>
<th></th>
<th>hamming</th>
<th>em</th>
<th>gcode</th>
<th>lambda</th>
<th>qsort</th>
<th>sched</th>
<th>wave</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard combinators</strong></td>
<td><strong>G-machine</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.42</td>
<td>2.99</td>
<td>6.68</td>
<td>4.27</td>
<td>6.97</td>
<td>11.29</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td><strong>G-machine</strong></td>
<td><strong>Flow graph</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.00</td>
<td>1.95</td>
<td>2.51</td>
<td>1.98</td>
<td>2.42</td>
<td>1.65</td>
<td>25.10</td>
<td></td>
</tr>
</tbody>
</table>

Table 2 presents the reduction in cell claims achieved by the G-machine over standard combinators and that achieved by the flow graph compiler over the G-machine. The wave program does not perform as well as the other programs on the G-machine, because it essentially requires full-lazy lambda lifting, [101] which the G-machine compiler does not provide. At present the flow graph compiler does not perform full-lazy lambda lifting either. Nevertheless the performance of the wave program with the flow graph compiler is back in line with the others. The reason is, that the two functions most often used by the wave program are passed suspensions as arguments by the G-machine. These functions are strict in all their arguments, which is detected and used to great advantage by the flow graph compiler. It generates code to evaluate the argument expressions eagerly and pass their values on the stack. Because the two functions are called so often, this alone accounts for a reduction of over 300000 cell claims.

## 7 Conclusions and Future Work

High performance compilers for lazy functional languages are beginning to come within reach as compiler writers discover the vast body of knowledge that the implementors of imperative
languages have gathered. We found the flow graph as the representation of a function being compiled a useful concept. Information flow from the place where the information is gathered to where it is needed is both intuitive and easy to formalise. We have shown that detecting use of arguments of a function can be expressed concisely and elegantly as a set of rewrite rules over a set of nodes from the flow graph. The intuition is supported by the possibility of drawing pictures of flow graphs and working out the placement of the appropriate property on each edge.

Expressing program analysis as a set of rewrite rules over a multi set allows for a simple, yet effective implementation to be made in a standard lazy functional language. Finding fix points in abstract interpretation is computation intensive and difficult to control. Yet the method of repeated approximation by rewriting multi sets provides a handle that makes it easy to cause the analysis and synthesis passes of our compiler without impairing the quality of the generated code. The medium size functional programs that we have compiled and run so far show a consistent performance improvement over our scalar version of the (1984) G-machine compiler. Several more well known optimisations need to be incorporated in the compiler (e.g. using vector apply nodes when possible [14]).

The prototype of our compiler is capable of translating programs into the functional benchmark that we have available. The next stage is to extend our measurements to a wider range of programs, collecting a greater variety of experimental data than the call chain rates above. We will also examine the effect of the various optimisations to assess their relative significance. Finally, the translation to machine code will allow us to investigate some of the optimisations (such as general tail calls) which are difficult to express in C, and provide the facility to compare absolute performance with other languages.

The authors would like to thank the many people who have helped in the development of this paper and the production of this paper, in particular Marcel Beemster, Andy Gravel, Koos Langendoen and Wim Vree.

References


