

A Pragmatic Approach to the Analysis and Compilation of Lazy Functional Languages

Hugh Glaser Pieter Hartel*
John Wild

Department of Electronics and Computer Science
University of Southampton

Abstract

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 describing the optimisations and code generation for such a compiler, which allows the exploitation of many standard and some new techniques 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.

1 Introduction

The FAST (Functional programming for ArrayS of Transputers) Project, funded by the UK government, 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 implementation 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 some preliminary results and conclusions.

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:

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

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.

1.2 Graph Reduction

Graph reduction [23] is an abstract machine for evaluating functional languages, maintaining the 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 figure 2.)

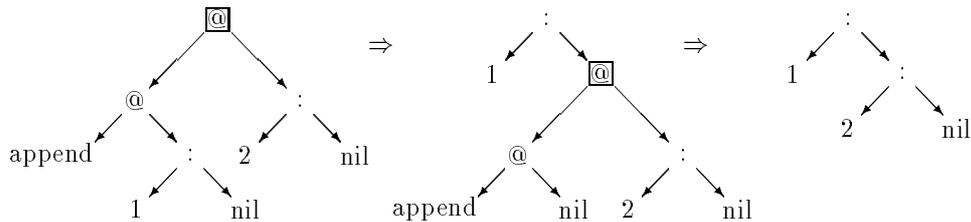


Figure 2: Reduction of *append* [1] [2] to yield [1,2].

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 list. (The application nodes that are selected for reduction and subsequently updated have been enclosed in boxes for identification.) There are a number of issues involved in correctly implementing graph reduction. For example the second argument to *append* is never examined, and should therefore not be evaluated if it is as yet unevaluated, and there is a need to update each 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 a two stroke 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 the created graph into native machine code, short-circuiting the graph reducer altogether. This has the added bonus that it reduces run time graph creation, which helps to avoid heap cell claims. 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. def/use 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 λ -calculus. The two standard axioms of the λ -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. Strictly speaking this is all that is required for a flow graph representation of the untyped λ -calculus. However a small number of extra primitives are highly desirable. These are SWITCH and MERGE to represent IF ... THEN ... ELSE ... FI; IMPORT and EXPORT to regulate the exchange of information across function boundaries; CALL and RESULT to present the compiler with information about the presence of fully parameterised calls, as opposed to partial applications, and fully built functions. The example of figure 3 shows the flow graph for the ubiquitous factorial function.

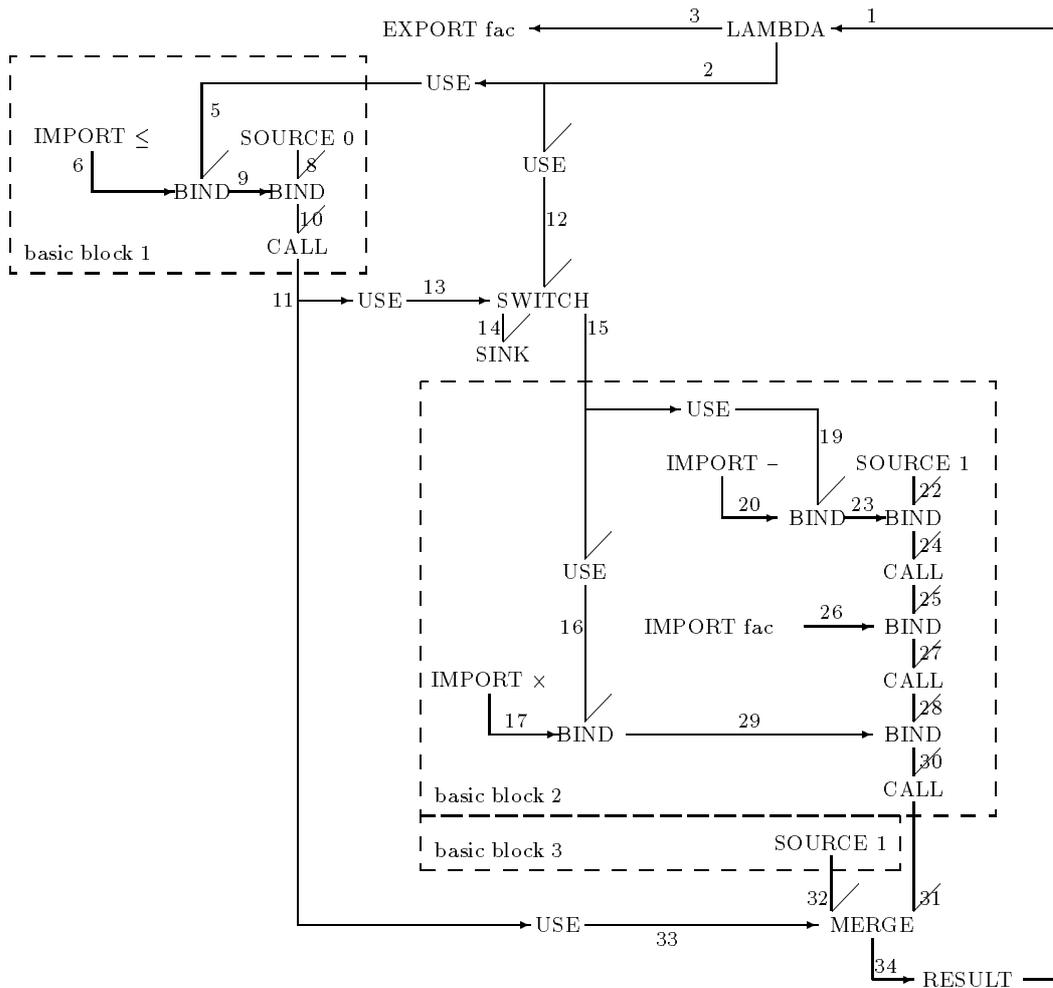


Figure 3: Flow graph of $fac = \lambda n. IF n \leq 0 THEN 1 ELSE n \times fac (n-1) FI$

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, ' \leq ', 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 ' \leq ' is known to be strict in its two inputs. This implies that to produce an output value, the function ' \leq ' 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 ' \leq ' 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 ' \times ' 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 of the program are not affected, but the efficiency is. In a strict context it is generally more expensive to build a suspension for an expression and delay its evaluation than it is to evaluate an expression immediately and pass its value around.

The data driven reading of a flow graph is perhaps a more intuitive one. Suppose for instance that factorial is called with an argument of 3. No matter how parameters are passed to functions (see below), the value 3 must enter the flow graph of figure 3 at the edge labelled 2. The value 3 is then copied by the two USE nodes to edges 5 and 12 (data values flow along the edges in the direction of the arrow heads). The purpose of the SWITCH node is to send the value of its top input (edge 12) to either its left (edge 14) or right output (edge 15), depending on the truth value on its control input (edge 13). The strict requirement tells us that, whatever basic block 1 contains, it must be evaluated before the parameter can be passed through the SWITCH node.

Returning to basic block 1 we evaluate the expression ‘ $3 \leq 0$ ’, and obtain the value FALSE on edge 11, which is subsequently used on edges 13 and 33. The SWITCH node therefore sends the value 3 from edge 12 to edge 15, where it can be manipulated further by the nodes of basic block 2. After evaluating basic block 2, the recursive calls to factorial will have completed, leaving a value 6 on edge 31. Note that the MERGE node must share the same control input as the corresponding SWITCH node, in order that the correct input (edge 31), is routed to the output (edge 34). The RESULT node finally sends this result back to the caller.

3 A More Formal Treatment of Flow Graphs

The 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 of compilation. Each function is represented as a sub-graph. The functions are connected via edges and USE nodes. All edges in a compilation unit are uniquely numbered. The nodes are represented as 4-tuples of the form $(inputs, type, qualifiers, outputs)$. Here, *inputs* and *outputs* are lists of edges, *type* indicates which of the 11 different node types the 4-tuple represents. Each node carries a list of further *qualifiers* that represent synthesised information pertinent to the node. Each phase of the analysis adds further elements to the set of names and modifies or removes existing elements as appropriate. The elements particular to a phase of the analysis are called attributes. This mechanism is loosely based on the Γ mode of computation [2]. An attribute is a triple of the form $(type, edge, value)$. The *type* specifies which attribute the *value* represents on the particular *edge*. For instance $(STR, 6, T)$ means that edge 6 carries the strict attribute (where $T = TRUE$, $F = FALSE$). Transformations on the multiset nodes and attributes are specified by means of rewrite rules with pattern matching on multiset elements or components thereof. The set of rewrite rules that deal with one particular attribute is called a compilation scheme. Consider the compilation scheme for the strict property as shown in figure 4.

The clauses of the function *Str* when applied to a multiset are tried top down, left to right on elements of the multiset. The words spelled in upper case are literals that must be present as 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

Figure 4: Compilation scheme for the “strict” property

1.	$\text{Str } \{a, \text{LAMBDA}, p, (l, c);$	$\text{STR}, l, sl; \text{STR}, c, sc; x\}$	$=$	$\text{Str } \{\text{STR}, a, sc;$	$x\} \cup \star\{-x\}$
2.	$\text{Str } \{(a, b), \text{BIND}, p, c;$	$\text{ABS}, a, (n, \{n+1, l, fs\}); \text{STR}, l, T; \text{STR}, c, T; x\}$	$=$	$\text{Str } \{\text{STR}, a, T; \text{STR}, b, T; \text{STR}, l, T;$	$x\} \cup \star\{-x\} - \{\text{STR}, l, T\}$
3.	$\text{Str } \{(a, b), \text{BIND}, p, c;$	$\text{STR}, c, sc; x\}$	$=$	$\text{Str } \{\text{STR}, a, sc; \text{STR}, b, F;$	$x\} \cup \star\{-x\}$
4.	$\text{Str } \{(a, b), \text{SWITCH}, p, (c, d);$	$\text{STR}, c, sc; \text{STR}, d, sd; x\}$	$=$	$\text{Str } \{\text{STR}, a, F; \text{STR}, b, sc \ \& \ sd;$	$x\} \cup \star\{-x\}$
5.	$\text{Str } \{(a, b, c), \text{MERGE}, p, d;$	$\text{STR}, d, sd; x\}$	$=$	$\text{Str } \{\text{STR}, a, sd; \text{STR}, b, sd; \text{STR}, c, sd;$	$x\} \cup \star\{-x\}$
6.	$\text{Str } \{, \text{SOURCE}, p, a;$	$\text{STR}, a, sa; x\}$	$=$	$\text{Str } \{$	$x\} \cup \star\{-x\}$
7.	$\text{Str } \{a, \text{SINK}, p, ;$	$x\}$	$=$	$\text{Str } \{\text{STR}, a, F;$	$x\} \cup \star\{-x\}$
8.	$\text{Str } \{, \text{IMPORT}, p, a;$	$\text{STR}, a, sa; x\}$	$=$	$\text{Str } \{$	$x\} \cup \star\{-x\}$
9.	$\text{Str } \{a, \text{EXPORT}, p, ;$	$x\}$	$=$	$\text{Str } \{\text{STR}, a, F;$	$x\} \cup \star\{-x\}$
10.	$\text{Str } \{a, \text{CALL}, p, b;$	$\text{STR}, b, sb; x\}$	$=$	$\text{Str } \{\text{STR}, a, sb;$	$x\} \cup \star\{-x\}$
11.	$\text{Str } \{a, \text{RESULT}, p, b;$	$x\}$	$=$	$\text{Str } \{\text{STR}, a, T;$	$x\} \cup \star\{-x\}$
12.	$\text{Str } \{a, \text{USE}, p1, b1; \dots; a, \text{USE}, pn, bn;$	$\text{STR}, b1, sb1; \dots; \text{STR}, bn, sbn; x\}$	$=$	$\text{Str } \{\text{STR}, a, sb1 \text{ --- } \dots \text{ --- } sbn;$	$x\} \cup \star\{-x\}$
13.	$\text{Str } x$		$=$	x	

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 $(a, LAMBDA, p, (l, c))$ and the attributes (STR, l, sl) and (STR, c, sc)) 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 *Str*: a represents the input edge of the LAMBDA node and l and c its output edges. p is the qualifier that applies to the LAMBDA node; sl and sc are the strictness values on edges l and c respectively. The variable x is bound to the the remaining elements of the multiset. We found it convenient to also use a symbol ‘ \star ’ that represents the multiset as a whole as a default binding. It could be made explicitly, in the case of clause 1 for example as: *WHERE* $\star = \{a, LAMBDA, p, (l, c); STR, l, sl; STR, c, sc; x\}$. The rewrite that takes place after the match of clause 1 has succeeded, generates a new attribute (STR, a, sc) for the input edge of the LAMBDA node. Its strictness value is merely a copy from the c -output of the LAMBDA node. After the rewrite, further matches are attempted on the union of the newly generated element $\{STR, a, sc\}$ and the multiset $\{x\}$. The remaining elements $\star - \{x\}$ 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 function argument bound by the BIND node is a strict argument. This uses an attribute called *ABS* (for abstraction). It records for each actual argument (b input edge to a BIND node) to which formal argument (l output edge of a LAMBDA node) it corresponds. The reason that there are two clauses present for the BIND node is that, depending on the order in which elements of the multiset that represents a module are processed by the *Str* function, the strictness of a formal argument may or may not yet be known before the information is required. The first BIND clause (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 to match all nodes from the multiset that specify the USE of a particular edge a . The qualifiers and output edges are bound to the variables $p1 \dots pn$ and $b1 \dots bn$ respectively. The variable n is bound to the number of USE nodes found, where n is a positive integer. The attribute for the edge a is the logical conjunction of $sb1 \dots sbn$ of the attributes on the output edges of all the USE nodes.

Rewriting goes on until none of the clauses 1 — 12 apply. The clause at line 13 then returns what 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 selected for matching. This has the advantage that the method is cheap to implement; its complexity is of $O(n)$ where n is the number of nodes in the flow graph. Since the strictness information is only an approximation, we can improve the information by re-applying the function *Str* to its own output, perhaps repeating this process several times. For the programs that we have analysed so far we found no further improvements after 4 runs of the analysis. Theoretically, plateaus present severe obstacles to the derivation of complete first order strictness in flat domains [4]. We are not particularly worried by this phenomenon, as the analysis so far allows us to experiment effectively with the problems of synthesis, and the system allows the import of strictness information from more powerful analysers. Pattern matching often causes functions to make tests on the form of their arguments, such that our simple strictness analyser discovers these arguments as strict. There are examples such as the one given by Clack et al [4]. that our analysis can not approximate well: *f x y = IF (x = 0) THEN y ELSE (f (x-1) y) FI*. Even though this function is strict in both arguments our analysis will only discover it to be strict in x .

4 The Implementation of a Compilation Scheme

The description of the schemes is in a convenient form for understanding, but pattern matching on large multisets is not a sensible implementation method. We manually translate the schemes into 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 numbers 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 example 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	
2,	USE,(),	12	2,	USE,(),	12	
3,	EXPORT,fac,	()	3,	EXPORT,fac,	()	
()	IMPORT,≤,	6	()	IMPORT,≤,	6	
(6,5),	BIND,(),	9	(6,5),	BIND,(),	9	
()	SOURCE,0,	8	()	SOURCE,0,	8	
(9,8),	BIND,(),	10	(9,8),	BIND,(),	10	
10,	CALL,(),	11	10,	CALL,(),	11	
11,	USE,(),	13	11,	USE,(),	13	
11,	USE,(),	33	11,	USE,(),	33	
(13,12),	SWITCH,	(14,15)	(13,12),	SWITCH,	(14,15)	
14,	SINK,(),	()	14,	SINK,(),	()	
15,	USE,(),	16	15,	USE,(),	16	
15,	USE,(),	19	15,	USE,(),	19	
()	IMPORT,×,	17	()	IMPORT,×,	17	
(17,16),	BIND,(),	29	(17,16),	BIND,(),	29	
()	IMPORT,-,	20	()	IMPORT,-,	20	
(20,19),	BIND,(),	23	(20,19),	BIND,(),	23	
()	SOURCE,1,	22	()	SOURCE,1,	22	
(23,22),	BIND,(),	24	(23,22),	BIND,(),	24	
24,	CALL,(),	25	24,	CALL,(),	25	
()	IMPORT,fac,	26	()	IMPORT,fac,	26	
(26,25),	BIND,(),	27	(26,25),	BIND,(),	27	
27,	CALL,(),	28	27,	CALL,(),	28	
(29,28),	BIND,(),	30	(29,28),	BIND,(),	30	
30,	CALL,(),	31	30,	CALL,(),	31	
()	SOURCE,1,	32	()	SOURCE,1,	32	
(33,32,31),	MERGE,	34	(33,32,31),	MERGE,	34;	STR,34,T
34,	RESULT,	1				

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 run time 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 *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: *fac* ($a - 1$). The imperative compiler generates code to subtract one from a and passes the difference to the recursive invocation of *fac*. No heap cells are allocated or even accessed throughout the execution of a call to *fac*.

```
int fac (a)
int a;
{
    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 *unwind* function to evaluate a suspension. Letting the the callee 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. *gcode* 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]

category	fib	hamming	em	gcode	lambda	qsort	sched	wave
<i>Standard combinators</i>								
total	289	15970	733742	966963	77128	1131047	477243	424623
<i>G-machine</i>								
apply cells	73	1604	182786	130889	14025	129508	33740	324745
cons	0	577	41328	10554	2486	23837	8014	1935
number	36	303	20894	3266	1547	8854	492	76965
total	109	2484	245008	144709	18058	162199	42246	403645
<i>Flow graph</i>								
apply cells	0	1588	79910	43822	6252	34101	18156	8209
cons	0	577	42730	9478	2124	23900	6399	1923
number	0	303	2461	4271	734	8855	930	5946
total	0	2468	125101	57571	9110	66856	25485	16078

Table 1: Cell claims for example programs

hamming	em	gcode	lambda	qsort	sched	wave
<i>Standard combinators/G-machine</i>						
6.42	2.99	6.68	4.27	6.97	11.29	1.05
<i>G-machine/Flow graph</i>						
1.00	1.95	2.51	1.98	2.42	1.65	25.10

Table 2: Reduction in cell claims for example programs

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 Turners 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 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 so well as the other programs on the G-machine, because it essentially

requires full-lazy lambda lifting, [10] 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 strict 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 throttle 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. This improvement can be attributed almost entirely to the extension of analyses across function boundaries. 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 the programs in 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 cell claim 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 work and production of this paper, in particular Marcel Beemster, Andy Gravell, Koen Langendoen and Wim Vree.

References

- [1] L. Augustsson, T. Johnsson, *The Chalmers Lazy-ML compiler*, "The Computer Journal", **32** (2) pp. 127-141, Feb. 1989.

- [2] J.-P. Banâtre, D. Le Metayer, *A new computational model and its discipline of programming*, “INRIA research report 566”, 1986.
- [3] G. L. Burn, S. L. Peyton Jones, J. D. Robson, *The spineless G-machine*, “ACM symp. on Lisp and functional programming”, Snowbird, Utah, ACM, pp. 244-258, Jul. 1988.
- [4] C. Clack, S. L. Peyton Jones, *Strictness analysis — a practical approach*, “Second conf. on functional programming languages and computer architecture, LNCS 201”, Nancy, France, J.-P. Jouannaud (eds), Springer Verlag, pp. 35-49, Sep. 1985.
- [5] J. Cohen, *Garbage collection of linked structures*, “Computing Surveys”, **13** (3) pp. 341-367, Sep. 1981.
- [6] S. Cox, H. Glaser, M. Reeve, *Compiling functional languages*, “Proc. of the workshop on implementation of functional languages”, Aspenas, Sweden, T. Johnsson, S. L. Peyton Jones, K. Karlsson (eds), Dept. of Comp. Sci, Chalmers Univ. of Technology, Goteborg, Sweden, Programming Methodology group report 53, pp. 145-156, Sep. 1988.
- [7] J. Fairbairn, S. Wray, *Tim: A simple lazy abstract machine to execute supercombinators*, “Third conf. on functional programming languages and computer architecture, LNCS 274”, Portland, Oregon, G. Kahn (ed), Springer Verlag, pp. 34-45, Sep. 1987.
- [8] P. H. Hartel, *A comparison of three garbage collection algorithms*, PRM project internal report D-23, Dept. of Comp. Sys, Univ. of Amsterdam, Feb. 1988.
- [9] P. H. Hartel, A. H. Veen, *Statistics on graph reduction of SASL programs*, “Software—Practice and Experience”, **18** (3) pp. 239-253, Mar. 1988.
- [10] P. H. Hartel, *Performance of lazy combinator graph reduction*, PRM project internal report D-27, Dept. of Comp. Sys, Univ. of Amsterdam, Aug. 1989.
- [11] P. Henderson, *Functional programming — Application and implementation*, Prentice Hall, 1980.
- [12] P. Hudak, P. Wadler (eds), *Report on the programming language Haskell — A non-strict purely functional language, Version 1.0*, Dept. of Comp. Sci, Univ. of Glasgow, UK, Apr. 1990.
- [13] R. J. M. Hughes, *Why functional programming matters*, “The Computer Journal”, **32** (6) pp. 89-107, Apr. 1989.
- [14] T. Johnsson, *Efficient compilation of lazy evaluation*, “ACM SIGPLAN 84 symp. on compiler construction”, “SIGPLAN notices”, **19** (6) pp. 58-69, Jun. 1984.
- [15] P. H. J. Kelly, *Functional programming for loosely-coupled multiprocessors*, Pitman publishing, 1989.
- [16] P. W. M. Koopman, *Interactive programs in a functional language: A functional implementation of an editor* Software—Practice and Experience, **17** (9), pp. 609-622, Sep. 1987.
- [17] S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice Hall, 1987.

- [18] D. A. Turner, *A new implementation technique for applicative languages*, “Software—Practice and Experience”, **9** (1) pp. 31-49, Jan. 1979.
- [19] D. A. Turner, *SASL language manual*, Computing Laboratory, Univ. of Kent at Canterbury, Technical report, Aug. 1979.
- [20] D. A. Turner, *Miranda: A non-strict functional language with polymorphic types*, “Second conf. on functional programming languages and computer architecture, LNCS 201”, Nancy, France, J.-P. Jouannaud (eds), Springer Verlag, pp. 1-16, Sep. 1985.
- [21] A. H. Veen, *Dataflow Machine Architecture*, “ACM Computing Surveys”, **18** (4) pp. 365-396, Dec. 1986.
- [22] W. G. Vree, *Design considerations for a parallel reduction machine* Ph.D. thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec. 1989.
- [23] C. P. Wadsworth, *Semantics and pragmatics of the lambda calculus*, Ph.D. thesis, Oxford Univ, U.K., 1971.
- [24] S. C. Wray, *A new strictness detection algorithm*, “Proc. of the workshop on implementation of functional languages”, Aspenas, Sweden, L. Augustsson, R. J. M. Hughes, T. Johnsson, K. Karlsson (eds), Dept. of Comp. Sci, Chalmers Univ. of Technology, Goteborg, Sweden, Programming Methodology group report 17, Feb. 1985.