Compilation of functional languages using
flow graph analysis

Pieter H. Hartel * Hugh Glaser † John M. Wild ‡
July 21, 1996

SUMMARY

A system based on the notion of a flow graph is used to formally specify and to implement a
compiler for a lazy functional language. The compiler takes a simple functional language as input
and generates C. The generated C program can then be compiled, and loaded with an extensive
run-time system to provide the facility to experiment with different analysis techniques. The
compiler provides a single, unified, efficient, formal framework for all the analysis and synthesis
phases, including the generation of C. Many of the standard techniques, such as strictness and
boxing analyses, have been included.

KEY WORDS

Compilation of lazy functional languages, flow graph, program analysis, program synthesis,
strictness.

INTRODUCTION

The functional programming community has identified a number of different analysis techniques
that can improve the implementation of functional languages. Each of these is of sufficient
complexity that it requires a significant body of research to exploit to the full. A good imple-
mentation, however, needs to perform a large number of such analyses, as well as synthesising
appropriate code to take advantage of the information provided. Fortunately, many of the anal-
yses have similar structures, and so a system can be provided in which analysis techniques can
be imported, provided they are defined in the required manner. In addition to providing a
framework for the analysis, it is desirable that the system should provide a similar framework
for the synthesis, ideally as far as the production of code.

As compile time analysis for functional programming languages becomes more complicated,
good compiler organisation becomes increasingly important. Our method, based on the flow
graph, focuses on compiler organisation as well as the need to generate efficient code. The
flow graph framework allows for importing most of the analyses that have been reported in the
literature into a single system. This is done in such a way that it is both possible to reason
about the optimisations in a formal way and to implement them efficiently.

*Dept. of Computer Systems, University of Amsterdam, Kruishan 403, 1098 SJ Amsterdam, The Netherlands,
Email: pieter@fwi.uva.nl
†Dept. of Electronics and Computer Science, University of Southampton, Southampton, SO9 5NH, England,
Email: H.Glaser@southampton.ac.uk
‡XAM Ltd, 71a Town street, Armley, Leeds, LS12 3HD, England
In the literature many program analyses have been proposed [1], but strictness analysis is the most popular in the functional programming community. We will therefore concentrate on strictness analysis in the examples. The analyses performed by the compiler include many others, such as boxing and sharing analysis and all have been specified in the same formal framework.

In the literature, a large variation is found in the abstract domain of strictness analysis. The more general the abstract domain, the more there are practical and to a lesser extent theoretical difficulties when solving the domain equations by fixed point finding techniques [2]. Our pilot compiler [3] performed nothing more advanced than backwards abstract interpretation on a two-point domain with a very simple fixed point finding algorithm and it showed promising results. Since then, both the domain and the fixed point finding algorithm have been extended.

From the outset we found that the abstract syntax tree when used as a basis for abstract interpretation methods is too limited. In the compiler the functions being compiled are represented as a flow graph, which is essentially an abstract syntax tree turned into a graph by tying the leafs of the syntax tree that represent function arguments back to the root. The structure thus formed supports forwards analysis as easily as backwards analysis. This is an advantage because it makes it possible to integrate the best kinds of analysis in a single framework. Examples of forwards and backwards analysis [4] will be given.

Attribute grammars [5, 6] can also be used to express forwards and backwards analysis, but attribute grammar systems are primarily concerned with specifying computations on syntax trees. Although there is a close relationship between a syntax tree and a flow graph, for the purpose of abstract interpretation of functional programs it is all-important to trace explicitly where variables, such as function arguments and names of let expressions, are defined and where they are used. By restricting the flow of information to that required for functional languages, a special purpose system is provided that is more efficient than a general purpose attribute grammar system. To express attribute propagation a notation is used that is more explicit (and also lower level) than that found in general purpose attribute grammar systems.

The next section gives an introduction to the implementation of lazy evaluation using compiled graph reduction. In the third and fourth sections, the use of a flow graph is discussed as a basic data structure to support the analysis and compilation of lazy functional programs. Section 3 shows that a flow graph allows for a theoretically sound underpinning of the compilation algorithms in the sense that a flow graph can be given the same meaning as the standard denotational semantics commonly defined for lazy functional languages. Section 4 presents two of the many analysis phases performed by the compiler in some detail, showing how a flow graph supports the intuition of what is going on during the analysis. The implementation of an analysis over a flow graph is described in Section 5. In Section 6 some of the results are compared to those reported by other researchers and the performance figures for both the compiler and the compiled code are discussed. The last section presents the conclusions.

AN INTRODUCTION TO THE IMPLEMENTATION OF LAZY EVALUATION

The evaluation strategy used in an implementation of a purely functional language such as Miranda [7] or Haskell [8] is lazy evaluation [9]. This strategy is radically different from that used in implementations of imperative programming languages such as C, Pascal and Fortran. Lazy evaluation determines, partly at compile time and partly at runtime, when expressions are evaluated; the programmer determines only what will be evaluated. In imperative languages, the programmer determines to a large extent when expressions are evaluated.

The lazy evaluation strategy is best explained using an example. Consider the function Append in Figure 1, which is written in a functional notation close to Miranda and Haskell.

---

1Miranda is a trademark of Research software Ltd.
Function application is indicated by juxtaposition. Parentheses are used to enforce priorities, not to gather the arguments to a function as is usual in many other programming languages. For example, \( (\text{Append} \ (Tl \ x) \ y) \) should be read as: apply \( Tl \) to \( x \), then apply \( \text{Append} \) to its two arguments, \( (Tl \ x) \) and \( y \).

1. \( \text{Append} : * \rightarrow * \rightarrow * \)
2. \( \text{Append} \ x \ y = \text{if} \ Null \ x \ \text{then} \ y \ \text{else} \ \text{Cons} \ (\text{Hd} \ x) \ (\text{Append} \ (Tl \ x) \ y) \)
3. \( \text{Main} = \text{Append} \ (\text{Cons} \ 1 \ \text{Nil}) \ (\text{Cons} \ 2 \ \text{Nil}) \)

Figure 1: Definition of running example \( \text{Append} \) and a main expression

The first line of the program is the type specification of \( \text{Append} \). The type defines \( \text{Append} \) to take two arguments, both lists of (the same) unspecified element type \( * \), and to return a list of the same type. The second line tests whether the first argument is an empty list. In that case, the \( \text{Null} \) test yields \( \text{True} \) and the second argument list is returned. Otherwise the constructor function \( \text{Cons} \) combines the first element of the present list with the recursive application of \( \text{Append} \) (line 3).

Most lazy functional languages provide elaborate syntax to allow for a more elegant formulation of functions using pattern matching on arguments and infix operators. As the purpose of this section is to explain the principles of the evaluation strategy we have refrained from using such syntax as it would unnecessarily complicate matters.

The main expression at line 4 in Figure 1 applies \( \text{Append} \) to 2 singleton lists. Here \( \text{Nil} \) represents the empty list. The answer of this program is thus a two element list, whose first element is 1, and whose second element is 2.

The example shows that:

- \( \text{Append} \) is a pure function: it has no side effects, as there are no assignments present;
- the order of evaluation is not explicitly stated (note that there are no semi colons);
- allocation and reclamation of storage is left to the system (garbage collection [10]);
- the interface to the outside world is completely specified by the parameters and result.

Functional programming takes the burden of much of the book-keeping associated with the actual computation away from the programmer. Functional languages thus have a number of advantages [11].

Unfortunately, there is a price to be paid for these advantages. In relieving the programmer of the burden of specifying evaluation order and storage management, the system requires greater intelligence on the part of the implementation. In practice, this means that the implementation should support a suitable execution model and have an efficient storage management system. We will say no more about storage management systems; a good survey is given by Cohen [10].

A good execution model for functional languages is lazy evaluation. The two key properties of lazy evaluation are: firstly, an expression is evaluated only when its value is needed, and secondly an expression is evaluated at most once. The implementation and not the programmer decides when expressions are evaluated. In particular, when the value of an expression is never needed, it will never be evaluated. Lazy evaluation can be implemented entirely at runtime, by interpreting a suitable representation of the program [12, 9]. Deciding at runtime which expressions need to be evaluated next is expensive; much can be gained by making decisions at
The purpose of strictness analysis. According to the classical definition, a monadic function of type \( f : d \to r \) is strict in its argument if:

\[
\forall x : (x = \bot) \to (f x = \bot)
\]

Here the symbol \( \bot \) may be read as "totally undefined". The operational interpretation of this definition is that it is safe to evaluate \( x \) to head normal form \([13]\) (to a value or data constructor) before, or in parallel to evaluating \( f \ x \), because when the computation of \( x \) diverges, that of \( f \ x \) also diverges. Hence termination is not affected by the choice of evaluation strategy. Since expressions do not have side effects in pure functional languages, the correctness of the answer is also not affected by the evaluation strategy. The definition of strictness can be generalised to functions of more than one argument, by requiring that the strictness of a particular argument must be totally independent of the values of the other arguments.

**Graph Reduction**

Graph reduction \([14]\) is a technique for implementing lazy evaluation. It forms the basis for many implementations of functional languages \([15, 16, 17, 18, 19]\). To explain the principles of graph reduction, consider the evaluation of \( \text{Append} \) with the arguments \((\text{Cons} \ 1 \ \text{Nil})\) and \((\text{Cons} \ 2 \ \text{Nil})\), as shown in Figure 2. Here a list is built out of \text{Cons} \ (for list CONstructor) and \text{Nil} nodes and a suspended function application is built out of \text{SUSP} nodes. More will be said shortly about the purpose of a suspension. The left descendant of \text{Cons} and \text{SUSP} nodes points at the first argument and the right descendant points at the second argument.

![Graph Reduction Diagram](image)

Figure 2: Reduction of \( \text{Append} \ (\text{Cons} \ 1 \ \text{Nil}) \ (\text{Cons} \ 2 \ \text{Nil}) \) to \( \text{Cons} \ 1 \ (\text{Cons} \ 2 \ \text{Nil}) \)

The graph that represents the application of \( \text{Append} \) to its arguments is reduced (= evaluated) in two stages, using the rules of the program, to the graph that represents the required list. There are three important issues involved in correctly implementing graph reduction:

- **When \( \text{Append} \) is called**, the first argument \( x \) is always examined by the \text{Null} test. Therefore, the argument must have been evaluated before the \text{Null} test takes place. The function \( \text{Append} \) is said to be strict in its first argument. The implementation can take advantage of this knowledge by making the compiled code evaluate the actual argument expression \textit{before} calling \( \text{Append} \). Other examples of strict functions are \( \text{Hd} \), \( \text{Tl} \) and \( \text{Null} \).

- **The second argument to \( \text{Append} \) is never examined**, and should therefore not be evaluated if it is as yet unevaluated. The second argument to \( \text{Append} \) is thus a non-strict argument. The compiler must arrange for the compiled code to construct a \textit{suspension} for any actual argument expression that is passed as a second argument to \( \text{Append} \). A suspension is a form of an expression that can be saved, such that it can be reduced later, when the value of the expression is actually required. Another example of non-strictness is provided by
the Cons function, which is non-strict in both its arguments. This is actually the reason why in the second frame of Figure 2 a suspension is built for Append. The suspension building and evaluation mechanism is thus an implementation consequence of the first key property of lazy evaluation.

- To implement the second key property of lazy evaluation it is necessary to update each suspension when it is evaluated, in case it represents an expression that is shared between other expressions.

Sometimes the value of a suspension is never required, so work may be saved by storing an expression as a suspension rather than by reducing it and storing the result value. However, building and reducing suspensions carries a significant cost, so the compiler should make sure that suspensions are only built when necessary. When laziness and thus suspensions are not required, the compiler must generate ordinary code as in imperative programs. The strictness analysis pass of the compiler serves this purpose and it is an important part of an efficient implementation of a lazy functional language. In the next section, when the implementation is discussed, this point will be explained in more detail.

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 is created, and secondly this graph is reduced. It is often possible to short-circuit this process by compiling native machine code that mimics the actions of creating and reducing a graph without actually allocating any heap storage. Efficiency is gained by avoiding graph creation, graph interpretation and allocation and reclaimation of heap storage. To perform this optimisation, it is necessary to analyse the program to determine when native machine code can be compiled and when a suspension must be made that can be reduced later.

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 [13].

Combining graph reduction with the evaluation of a C program

To expose the principles of graph reduction when coupled with the standard call-by-value evaluation method of ordinary C programs, let us look at a complete C program that implements lazy evaluation of the Append function and the main expression given with it in Figure 1. The
C program will be presented in small units, some of which form part of the runtime library associated with the compiler, while other parts are generated by the compiler. The C program is based on actual compiled code and on the runtime system that has been developed with the compiler, but it has been simplified to make the presentation succinct.

Runtime data structures

The run time system manipulates tagged heap nodes. Four tag values are defined for the purpose of presenting the compiled code of the `Append` function:

```c
#define INT 1
#define CONS 2
#define NIL 3
#define SUSP 4
```

The heap consists of data structures of type `node`. Each node accommodates either an integer (tag `INT`), a list constructor (tag `CONS`), a nil node (tag `NIL`) or a suspension (tag `SUSP`). A suspension implements a suspended function application; in addition to the tag, it contains three fields: The first field (`fun`) points at a prelude routine (see below). The remaining two fields contain pointers to the arguments of the suspended function. A suspension must be capable of accommodating a pointer to a prelude routine and all pointers to its arguments. In the present example only binary functions such as `Append` can be handled. This restriction is not essential, but imposed to keep the presentation reasonably simple.

Each node has a tag field that allows the type of the node to be determined at run time. In some implementations of lazy functional languages, no tag is required. Instead the purpose of the tag is served in a different way, which requires less space but is also more complicated to explain [20].

```c
#define PTR struct NODE */
typedef struct NODE {
    int tag;
    union {
        int val; /* tag INT */
        struct {
            PTR hd; /* tag CONS */
            PTR tl;
        } cons;
        struct {
            PTR (*fun)(); /* tag SUSP */
            PTR arg1;
            PTR arg2;
        } susp;
    } data;
} node;
typedef PTR ptr;
```

Some macros are useful for accessing often used fields of the various data structures. These macros correspond to some of the primitive functions used in the example program:

```c
#define Null(x) ((x)->tag == NIL)
#define Hd(x) ((x)->data.cons.hd)
#define Tl(x) ((x)->data.cons.tl)
```
The heap and the allocation routines

The heap is defined as an array of nodes. To allocate a node from the heap, the variable top must be incremented. All nodes are thus of the same size, which is rather a waste of space. In a real implementation, nodes will be more tightly packed, but this has not been done to avoid cluttering the code with address and size calculations. Also, to keep the presentation simple, there is no provision for dealing with heap overflow. In a real implementation, this would trigger the garbage collector, which gathers the nodes that are still in use, and makes the remaining nodes available for reuse [10].

#define HEAP_SIZE 10000
node heap[HEAP_SIZE];
ptr top = heap;

For each node type, an allocation routine is defined, which claims a node from the heap, and initialises the fields of the node. The allocation routines increment the heap pointer top as a side effect. All four allocation routines return a pointer to the newly allocated node. The allocation routines call a trace routine, which prints the string passed as an argument and the contents of the allocated node. This allows a trace to be made of the reduction process when the program is executed. Tracing is not normally part of an implementation; it is included in the C code for demonstration purposes only.

ptr Int(i) { int i; { ptr r; r = top++; r->tag = INT; r->data.val = i; trace(3,"Int ","r",r); return r; }

ptr Nil() { ptr r; r = top++; r->tag = NIL; trace(3,"Nil ","r",r); return r; }

ptr Cons(h,t) { ptr h,t; { ptr r; r = top++; r->tag = CONS; r->data.cons.hd = h; r->data.cons.tl = t; trace(3,"Cons ","r",r); return r; }

ptr Suspend(f,a1,a2) { ptr (* f)(); ptr a1,a2; { ptr r; r = top++; r->tag = SUSP; r->data.susp.fun = f; r->data.susp.arg1 = a1; r->data.susp.arg2 = a2; trace(3,"Susp ","r",r); return r; }

This completes the discussion of the basic runtime support for graph reduction. There are two more routines to be defined (Reduce and Print), but we will first discuss the code generated by the compiler for Append and its main expression.

The C code generated for Append

The C code generated by the compiler for Append is shown in Figure 3. It consists of two parts. The first part is the routine Append, which is the genuine translation of Append into

7
C. The C routine has a structure that is very similar to that of Append as shown in Figure 1. The main difference (apart from the syntax) is that the C version contains calls to Reduce and Suspend, which together implement the required laziness. Where laziness is not required, a direct translation has taken place, for instance the boolean expression in the functional version is identical to that in the C version (again apart from the syntax). The compiler has worked hard to generate this particular code, because most of the laziness that is implied in the functional notation is not present in the C code.

```c
ptr Prel_Append();
ptr Reduce();

ptr Append(x,y)
ptr x,y;
{
  trace(5,"Append","x",x,"y",y);
  if (Null(x)) {
    return Reduce(y);
  } else {
    return Cons(Hd(x),Suspend(Prel_Append,Tl(x),y));
  }
}

ptr Prel_Append(x,y)
ptr x,y;
{
  trace(5,"Prel ","x",x,"y",y);
  return Append(Reduce(x),y);
}
```

Figure 3: The C code generated for the function Append.

The second part of the code generated for Append is the prelude routine Prel_Append. This routine can be viewed as a second “entry point” to Append. The only difference between the two is that, whenever Append is called, the first argument must already be evaluated to head normal form. Under circumstances where this can not be guaranteed, the compiler arranges for a call to Prel_Append instead. The compiled code for the recursive call in the body of Append uses Prel_Append instead of Append, because the compiler does not know whether Tl(x) will deliver a head normal form or not. In the prelude routine, the compiler generates a direct call to Append.

The reducer

The heart of the graph reduction machinery consists of two routines: the reducer (Reduce) and the need to print (Print). The reducer is called from many places to assure that a certain node is to head normal form. There is only one node type that does not qualify as a head normal form: the suspension. The reducer performs its task of activating a function that has been suspended as follows. First the pointer to the prelude routine is extracted from the suspension. This prelude routine is then called with the pointers to the arguments of the suspended function. The prelude routine calls the reducer to bring all strict arguments in head normal form (in the case of the Append example the reducer is applied to the first argument only), then calls the function proper.
As soon as the activated function returns, the prelude routine will also return. Then the suspension that used to be associated with the suspension is updated with the result that has just been computed. This is necessary to guarantee that no shared computation is evaluated more than once.

```c
ptr Reduce(x)
ptr x;
{
    ptr r;
    trace(3,"Reduce","x",x);
    switch (x->tag) {
        case SUSP:
            r = x->data.susp.fun(x->data.susp.arg1,x->data.susp.arg2);
            trace(5,"Update","x",x,"r",r);
            *x = *r;  /* Update the result */
            break;
        default:
            break;
    }
    return x;
}
```

The code shows that when the reducer is applied to a data object, it has no effect. One of the optimisations a good compiler applies is to remove redundant calls to the reducer.

The need to print

The execution of a lazy functional program is driven by the "need to print" the answer. The answer is often a list of some values. In the Append example, the result is a list of two integers, namely (Cons 1 (Cons 2 Nil)).

The need to print is implemented by the Print routine. It recurses through the data generated by the main program. The need to print ensures, that each node it encounters is in head normal form by calling the reducer on the node before examining its tag. The three cases in the code below cover all possible tag values that represent head normal forms:

```c
ptr Print(x)
ptr x;
{
    x = Reduce(x);
    switch (x->tag) {
        case INT:
            printf("%d",x->data.val);
            break;
        case NIL:
            break;
        case CONS:
            Print(x->data.cons.hd);
            Print(x->data.cons tl);
            break;
    }
}
```

The reducer and the need to print are part of the runtime system. The only missing part of the C program is a main function. This is generated by the compiler from the functional program.
The main program and the execution trace

The main program first constructs the two lists that will be appended and calls the Print routine to display the result list:

```c
main()
{
    ptr x,y,z;
    x = Cons(Int(1),Nil());
    y = Cons(Int(2),Nil());
    z = Suspend(Prel_Append,x,y);
    Print(z);
}
```

This completes the C program that implements the functional program from Figure 1.

To study the runtime behaviour of the program, the trace that it produces will now be examined. The trace consists of one line per routine call. The name of the routine being traced appears first, followed by a colon. Then for each argument of the routine, the address of the argument is printed, followed by an interpretation of the contents of the node. The interpretation begins with the name of the tag, and is followed by a printout of the fields of the node. All addresses are printed in hexadecimal, and the first heap address is 00.

The first six lines are the result of constructing the two argument lists by the main program:

- `Int : r=00(INT 1)`
- `Nil : r=10(NIL)`
- `Cons : r=20(CONS 00 10)`
- `Int : r=30(INT 2)`
- `Nil : r=40(NIL)`
- `Cons : r=50(CONS 30 40)`

The next line is produced by the call to Suspend also from the main program. Hereafter, the state of the graph in the heap is exactly as shown in the first frame in Figure 2.

- `Susp : r=60(SUSP Prel_Append 20 50)`

The next trace line is produced by the reducer on demand from the need to print. The reducer discovers a suspension at address 60 and thus calls the prelude routine Prel_Append:

- `Reduce: x=60(SUSP Prel_Append 20 50)`
- `Prel : x=20(CONS 00 10) y=50(CONS 30 40)`

The last trace line above, from the prelude routine, shows that the actual arguments are both head normal forms, so that in this case the call to reduce the first argument (below) is redundant. After discovering this, reducer returns control to Prel_Append:

- `Reduce: x=20(CONS 00 10)`

Having ensured that the first argument is a head normal form, Append is called. This is the last action taken by Prel_Append. The next trace line shows that Append is entered:

- `Append: x=20(CONS 00 10) y=50(CONS 30 40)`

The Append routine now tests its first argument using the macro Null. The tests yields False, so that a suspension is made of the recursive call to Prel_Append. The Append routine has now completed its task of creating a head normal form in the form of a list constructor and thus returns control to the reducer.
The reducer takes the result of the computation that has just been completed (at address 80) and copies it over the suspension at address 60:

\[ \text{Update: x=60(SUSP Prel Append 20 50) r=80(CONS 00 70)} \]

The need to print now regains control and recurses on the head of the new list constructor node at address 60. The need to print calls the reducer to make sure that the head of the list is a head normal form, which it is (the integer 1):

\[ \text{Reduce: x=00(INT 1)} \]

The integer is then printed:

1

At this point, the state of the heap is exactly as shown in the second frame of Figure 2. The need to print now calls the reducer to deal with the suspension at address 70. This proceeds along the same lines as the reduction of the SUSP node at address 60:

\[ \text{Reduce: x=70(SUSP Prel Append 10 50)} \]
\[ \text{Prel : x=10(NIL)} \quad y=50(CONS 30 40) \]
\[ \text{Reduce: x=10(NIL)} \]
\[ \text{Append: x=10(NIL)} \quad y=50(CONS 30 40) \]

When Append is entered for the second time, the end of the input list has been reached, so that this time the Null test yields True. The Append routine ensures that the result that it will return is indeed a head normal form:

\[ \text{Reduce: x=50(CONS 30 40)} \]

The second suspension of Append at address 70 is then updated with the constructor node at address 50:

\[ \text{Update: x=70(SUSP Prel Append 10 50) r=50(CONS 30 40)} \]

Finally the need to print receives the updated list constructor node and prints its contents:

\[ \text{Reduce: x=30(INT 2)} \]
\[ 2 \]
\[ \text{Reduce: x=40(NIL)} \]

During this discussion we have glossed over several important issues, many of which are necessary to make the mechanism efficient. The main point is that the implementation of lazy evaluation requires an optimising compiler, which is able to spot as many expressions as possible that do not require the laziness. Such expressions may be compiled using the ordinary C evaluation strategy, which is more efficient than the graph reduction machinery. The remaining sections of the paper discuss the core of such an optimising compiler, emphasising both the structure of the compiler and the analyses that avoid laziness whenever possible.

**THE FLOW GRAPH**

A flow graph is a compile time representation of a program. This section defines the definition of a flow graph, then defines a mapping from an abstract syntax onto a flow graph and finally gives a denotational semantics of a flow graph.
The definition of a flow graph

A flow graph represents a functional program that is being compiled. It is worth commenting that this graph is not the same as the graph that will be reduced at run time. A flow graph is more like an abstract syntax tree, with an important difference: a flow graph has extra structure that makes it more convenient for program analysis and synthesis than a syntax tree. This extra structure also makes it a graph. Figure 4 shows the flow graph for the function Append. The “extra” structure consists of the two SWITCH nodes, the two LAMBDA nodes and the connecting USE nodes. Removing these nodes would yield an abstract syntax tree.

The edges of a flow graph are labelled with an integer number. The nodes have names that indicate their purpose. There are 11 different node types. The BIND and LAMBDA nodes represent function application and abstraction respectively. A primitive data object is created by a SOURCE node and unwanted objects are explicitly destroyed by a SINK node. The IMPORT and EXPORT nodes regulate the exchange of information across function boundaries. The USE nodes specify that a particular object may be used at more than one place. This is achieved by allowing two or more USE nodes to connect to a common input edge. None of the other nodes in the flow graph are allowed to share an input edge or to share an output edge.

Although these seven nodes are sufficient to compile efficiently, adding a small number of node types enhances the ease of expressing the compilation algorithms without unduly increasing the complexity of the compiler. The extra nodes are SWITCH and MERGE to represent the if then else construct and CALL and RESULT to present the compiler with information about the presence of fully saturated calls (as opposed to partial applications) and fully abstracted functions.

Much of the program analysis and code synthesis in a compiler is based on value representations. It is thus convenient to keep explicit track of the generators and consumers of each value (cf. def/use information in conventional compilers). This is specifically the task of the SWITCH and USE nodes, which explicitly route information from where it is produced to where it is needed. These nodes also account for a major part of the extra information that makes a flow graph different from a syntax tree. The ideas involved are perhaps best exposed by giving an operational reading of what may happen when we try to “execute” the flow graph. Note that following the analyses, the compiler is capable of collapsing parts of the flow graph to straight sections of imperative code, while other parts will build suspensions.

When Append is called with two actual arguments, the first enters the graph at edge 104 and the second at edge 102. Following the direction of the arrows, the first argument is distributed by two USE nodes to edges 108 and 114. The BIND node binds the Null test function appearing on edge 109 to its input on edge 108 and produces a suspension on edge 110. The CALL node signals that this suspension is actually a fully saturated call, which could be evaluated if necessary. The data item flowing along edge 108 may either be a list that has been evaluated to a certain extent, or an as yet completely unevaluated object. The Null test, when asked to produce a truth value, requires its argument to be in head normal form, so a completely unevaluated object will have to be reduced to head normal form before the Null test can be carried out. The analyses to be discussed later decide when the object eventually appearing on edge 108 will be evaluated and also when the Null test function will be called.

We would like to stress that, in a flow graph, suspensions of functions may flow down arcs as well as ordinary data values and so the flow graph itself is neutral with respect to the order of evaluation.

From strictness analysis it follows that Append is strict in its first argument, so that the Null test is duly applied to a head normal form. The Null test causes a truth value rather than a suspension to emerge on edge 113, and also on edges 115, 119 and 156 (because of the three USE nodes).

Supposing the Null test yields False, the switch nodes will pass their input values on edges
Figure 4: Flow graph for $\text{Append } x \ y = \text{if Null } x \ \text{then y else Cons (Hd } x) \ (\text{Append (Tl } x) \ y)$
114 and 118 respectively down to edges 117 and 121 respectively. No values then appear on edges 116 and 120. After passing a few more BIND and CALL nodes, an object appears on edge 152, which represents the result of the current call to Append. Because the MERGE node shares its control input edge 156 with the two SWITCH nodes, MERGE will select input from edge 152 and pass that object to edge 159. Input 154, which does not carry a value, is ignored by MERGE. Finally the function result must be communicated to the caller by RETURN.

Should the Null test return True, the first argument to Append is disposed of by the SINK node at edge 116 and the second argument is passed through the USE node connecting edges 120 and 154.

A graphical representation is convenient to reason about in a loose manner, but does not lend itself to a more formal approach. Therefore the flow graph of all functions within a program to be compiled are manipulated as a set of nodes. A node is a 4-tuple of the following form:

\[ \text{input}(s), \text{type}, \text{qualifier}, \text{output}(s) \]

Here, \text{input}(s) and \text{output}(s) are tuples of edges. Depending on the type of the node, such a tuple contains 0, 1, 2 or 3 edges. The next component in the flow graph node is the type, which indicates one of the 11 node types that are defined. Each node carries a further qualifier that represents information specific to the node. The SOURCE node carries the sourced value as qualifier and the IMPORT and EXPORT export nodes carry the name of the imported or exported function. The qualifier of the remaining nodes is empty.

All edges in the flow graph of a compilation unit are uniquely numbered. In Figure 4 the edges of all but the LAMBDA and EXPORT nodes are read from left to right as they appear in the input and output tuples. The edges of LAMBDA and EXPORT are read from right to left.

The results of an analysis on a flow graph are represented by attributes of the edges. An attribute is a 3-tuple of the form:

\[ \text{attribute}, \text{edge}, \text{value} \]

The first component is the name of the attribute, the second the edge number to which it applies and the third the attribute value. All program analysis and synthesis of the compiler is specified as a function over the set in which nodes (4-tuples) and attributes (3-tuples) coexist. This set is called a flow graph set.

Here, as an example, is the subset of nodes that corresponds to the Null test in Append:

\[ \emptyset, \text{IMPORT}, \text{Null}, 109 \]
\[ \downarrow \]
\[ (109, 108), \text{BIND}, \emptyset, 110 \]
\[ \downarrow \]
\[ 110, \text{CALL}, \emptyset, 113 \]

The subset of nodes corresponding to the conditional expression in Append:

\[ \cdots = \text{if} \ \cdots \ \text{then} \ \cdots \ \text{else} \ \cdots \]

is the set:

\[ (156, 154, 152), \text{MERGE}, \emptyset, 159 \]
\[ \downarrow \]
\[ 159, \text{RETURN}, \emptyset, 101 \]

Gathering all the nodes in Figure 4 in a set yields the complete flow graph set for Append. A formal derivation of the set will be discussed in the following sections.
Intermediate, the compiler input language

The input language of the compiler, Intermediate, is intended as a target for a front end. Intermediate can be regarded as a form of sugared λ-calculus. Figure 5 shows the abstract syntax of an Intermediate program $p$. More complex constructs, such as pattern matching and local function definitions are not supported and must therefore be compiled to the syntax of Figure 5, prior to the transformation of the parse tree into a flow graph. Compilation of pattern matching, lambda lifting and several other standard transformations are more convenient to apply to a parse tree than to a flow graph. Peyton Jones [13] discusses these transformations in detail.

\[
p ::= d_1 \ldots d_n
\]
\[
d ::= f \ x_1 \ldots x_n = e \quad (f \text{ and } x_1 \ldots x_n \text{ are identifiers})
\]
\[
e ::= \text{let } x_1 = e_1 \ldots x_n = e_n \text{ in } e
\]
\[
| \quad \text{if } e_0 \text{ then } e_t \text{ else } e_e
\]
\[
| \quad e_f \ e_a \quad \text{(function application)}
\]
\[
| \quad x \quad \text{(name of an argument or let introduced name)}
\]
\[
| \quad f \quad \text{(function name)}
\]
\[
| \quad \text{Nil} \quad \text{(Nil or other constants)}
\]

Figure 5: The abstract syntax of Intermediate, the flow graph compiler input language

Intermediate does not support case and letrec expressions, because the simpler the input language, the fewer different node types are needed in the flow graph representation. The if construct could be replaced by a case construct, which would require multiway instead of binary switch and merge nodes. This is certainly feasible but not strictly necessary.

It is not possible to build a letrec construct into Intermediate, as this would introduce arbitrary cycles in the flow graph. This is not a restriction on the power of Intermediate, as programmers often write letrec expressions when a cascade of let expressions would do equally well. Prior to the generation of the flow graph, the compiler must translate all non-recursive letrec expressions into let expressions and it must transform the remaining, genuinely self referential local definitions into applications of the fixed point combinator Fix [21].

Set notation

Before we can describe how a flow graph may be created and used, we must digress briefly to introduce suitable notation. This consists of a notation for sets and for rewriting sets with pattern matching on suitably selected subsets. The notation is similar to that used in functional programming languages for pattern matching on lists. Braces (\{ \}) and semi-colons (;) are used with sets, in the same way as brackets ([ ]) and colons (:) are conventionally used with lists. Parentheses are used to group expressions as normal. The semantics of pattern matching on sets and lists differ in that the former do not imply any ordering on the elements, whereas the latter do. The two function definitions below illustrate these points. PairList always takes the first two elements of its argument list. On the other hand, when PairSet is applied to a set of numbers, any two set elements may be selected and added, followed by applying PairList to the remainder of the set:

\[
\text{PairList} : [\text{Num}] \rightarrow [\text{Num}] \quad \text{PairSet} : \{\text{Num}\} \rightarrow \{\text{Num}\}
\]
\[
\text{PairList} \ (a : b : x) = a + b : \text{PairList} x \quad \text{PairSet} \ (a ; b ; x) = a + b ; \text{PairSet} x
\]
\[
\text{PairList} \ x = x \quad \text{PairSet} \ x = x
\]

When pattern matching on either sets or lists, actual arguments are considered top down, left to right. Variables in patterns always consist of lower case letters, possibly adorned with subscripts.
These variables are bound to values by the pattern matching. Repeated variables of the same name must all match the same value.

Elements of tuples are separated by commas, which have a higher priority than the semi-colons that separate set elements. Thus when dealing with sets of tuples, the brackets surrounding tuples are omitted. A set containing three 2-tuples is written as: \{a, b; c, d; e, f\}. The function Zip below rewrites all pairs of 2-tuples in the argument set to 2-tuples of two element sets. Both \(x\) and (Zip \(x\)) are thus sets:

\[
\begin{align*}
\text{Zip} : & \quad \{ \ast \times \ast \} \rightarrow \{ \ast \} \times \{ \ast \} \\
\text{Zip} (a_1, b_1; a_2, b_2; x) & = \{a_1; a_2\}, \{b_1; b_2\}; \text{Zip } x \\
\text{Zip} x & = x
\end{align*}
\]

A shorthand notation is often convenient to refer to the entire argument set on the left hand side of a defining equation. The section symbol \(\S\) is used for this purpose. Define an auxiliary function \(\text{Aux}\) thus:

\[
\begin{align*}
\text{Aux} : & \quad \{ \ast \times \ast \} \rightarrow \{ \ast \} \times \{ \ast \} \\
\text{Aux} (a_1, b_1; a_2, b_2) & = \{a_1; a_2\}, \{b_1; b_2\}
\end{align*}
\]

The first clause of Zip can now be defined as follows:

\[
\text{Zip} (a_1, b_1; a_2, b_2; x) = \text{Aux} (\S \setminus x); \text{Zip } x
\]

Here \(\S\) is the asymmetric set difference operator.

It is also convenient to use the “…” notation to indicate that any number of the surrounding indexed patterns may occur:

\[
\begin{align*}
\text{Aux}^{'} : & \quad \{ \ast \times \ast \} \rightarrow \{ \ast \} \times \{ \ast \} \\
\text{Aux}^{'} (a_1, b_1; \ldots ; a_n, b_n) & = \{a_1 ; \ldots ; a_n\}, \{b_1 ; \ldots ; b_n\}
\end{align*}
\]

The function \(\text{Aux}^{'}\) rewrites any number of pairs in one step rather than exactly two as \(\text{Aux}\) does. The index pattern \(n\) is also bound to the appropriate actual value. If the pattern \(n\) is indeed a variable, there are no restrictions on the number of occurrences of \(a_i, b_i\). In particular there may be no occurrences at all, in which case \(n\) is bound to 0, and the interpretation of the definition of \(\text{Aux}^{'}\) becomes:

\[
\text{Aux}^{'} () = 0, 0
\]

Here 0 is the same as \(\{\}\), the empty set.

**Generating a flow graph from Intermediate**

The compiler constructs a flow graph from the parse tree of an Intermediate program using the compilation functions \(\text{Fprg}\), \(\text{Fdef}\) and \(\text{Fexp}\) shown in Figure 6. The presentation has been simplified where the code to calculate edge labels has been replaced by comments stating which variables are to be bound to new unique integer labels.

The function \(\text{Fdef}\) transforms a function definition into a flow graph. It generates the \text{LAMBDA}, \text{RETURN} and \text{EXPORT} nodes that are associated with the header of the function and relies on \(\text{Fexp}\) to generate the nodes for the body of the function. \(\text{Fexp}\) receives as parameters the expression to be compiled, an environment and an edge label. The environment initially maps the functions arguments onto edge labels. When encountering \text{let} expressions, the mapping is extended with the local symbols and the associated edge labels. The environment mapping is represented as a set of \((\text{name, edge})\) pairs. The edge label passed to \(\text{Fexp}\) as the third parameter is the output edge to be associated with the flow graph compiled for the expression.
\[ F_{prg} \text{[} d_1 \ldots d_n \text{]} = F_{def} \text{[} d_1 \text{]} \cup \ldots \cup F_{def} \text{[} d_n \text{]} \]

\[ F_{def} \text{[} f \ x_1 \ldots x_n = e \text{]} = l_0, \text{return}, 0, l_1; \]
\[ l_1, \text{lambda}, 0, (h_1, l_2); \ldots; l_n, \text{lambda}, 0, (h_1, l_{n+1}); \]
\[ l_{n+1}, \text{export}, f, 0; \]
\[ F_{exp} \text{[} e \text{]} \{ x_1, h_1; \ldots; x_n, h_n \} l_0 \]
where
\[ l_0 \ldots l_{n+1} \text{ and } h_1 \ldots h_n \text{ are unique labels} \]

\[ F_{exp} \text{[} \text{if b then else e} \text{]} \{ x_1, h_1; \ldots; x_n, h_n \} l_x \]
\[ = l_b, \text{use}, 0, l_m; (l_m, t, e), \text{merge}, 0, l_x; \]
\[ F_{exp} \text{[} b \text{]} \{ x_1, h_1; \ldots; x_n, h_n \} l_b \cup \]
\[ F_{swt} h_1 c_1 v_1 t_1 e_1 \cup \ldots \cup F_{swt} h_n c_n v_n t_n e_n \cup \]
\[ F_{exp} \text{[} t \text{]} \{ x_1, t_1; \ldots; x_n, t_n \} l_t \cup \]
\[ F_{exp} \text{[} e \text{]} \{ x_1, e_1; \ldots; x_n, e_n \} l_e \]
where
\[ F_{swt} h_i c_i v_i t_i e_i = l_b, \text{use}, 0, c_i; h_i, \text{use}, 0, v_i; (c_i, v_i), \text{switch}, 0, (t_i, e_i) \]
\[ c_1 \ldots c_n, v_1 \ldots v_n, t_1 \ldots t_n, e_1 \ldots e_n, \text{ and } l_b, l_t, l_e, l_m \text{ are unique labels} \]

\[ F_{exp} \text{[} \text{let x} = e_1 \ldots x_n = e_n \text{ in e} \text{]} r \ l_x \]
\[ = F_{exp} \text{[} e_1 \text{]} r \ l_1 \cup \ldots \cup F_{exp} \text{[} e_n \text{]} r \ l_n \cup \]
\[ F_{exp} \text{[} e \text{]} \{ x_1, l_1; \ldots; x_n, l_n; r' \} l_x \]
where
\[ r' = \{(y, h) \in r \mid y \neq x_i \wedge 1 \leq i \leq n\} \]
\[ l_1 \ldots l_n \text{ are unique labels} \]

\[ F_{exp} \text{[} f \ a \text{]} r \ l_x \]
\[ = (l_f, l_a), \text{bind}, 0, l_x; F_{exp} \text{[} f \text{]} r \ l_f \cup F_{exp} \text{[} a \text{]} r \ l_a \]
where
\[ l_f \text{ and } l_a \text{ are unique labels} \]

\[ F_{exp} \text{[} x \text{]} (x, h; r) \ l_x = h, \text{use}, 0, l_x \]
\[ F_{exp} \text{[} f \text{]} r \ l_x = 0, \text{import}, f, l_x \]
\[ F_{exp} \text{[} \text{true} \text{]} r \ l_x = 0, \text{source}, \text{true}, l_x \]
(Ture and other constants)

Figure 6: Simplified scheme to transform an Intermediate program into a flow graph (the call and sink nodes are added later by separate schemes not shown here)
The derivation of the flow graph for `Append` commences as shown below. The nodes that are generated in previous steps are not shown in subsequent steps; dots are shown instead:

\[
\begin{align*}
F\text{def} \text{ } \& \text{ } \text{ } & \text{if } \text{ } \text{Null } x \text{ } \text{then } y \text{ } \text{else } \text{Cons } (\text{Hd } x) \text{ } (\text{Append } (\text{Tl } x) \text{ } y) \\
& \text{=}
\end{align*}
\]

(\text{Fdef})

\[
159, \text{return}, \emptyset, 101; \\
101, \text{lambda}, \emptyset, (102,103); \text{ } 103, \text{lambda}, \emptyset, (104,105); \\
\text{ } 105, \text{export}, \text{Append}, \emptyset;
\]

\[
\begin{align*}
\text{Fexp} \text{ } \& \text{ } \text{ } & \text{if } \text{ } \text{Null } x \text{ } \text{then } y \text{ } \text{else } \text{Cons } (\text{Hd } x) \text{ } (\text{Append } (\text{Tl } x) \text{ } y) \\
& \text{=}
\end{align*}
\]

(\text{Fexp})

\[
\begin{align*}
\text{...}
& \text{113,use}, \emptyset, 156; \text{ } (156,154,152), \text{merge}, \emptyset, 159; \\
\text{Fexp} & \text{ } \text{ } \text{ } [ \text{Null } x ] \text{ } \{ x, 104; \text{ } y, 102 \} \text{ } 113 \cup \\
\text{Fexp} & \text{ } \text{ } \text{ } [ x ] \text{ } \{ x, 104; \text{ } y, 102 \} \text{ } 109 \cup \\
\text{Fexp} & \text{ } \text{ } \text{ } \{ x, 116; \text{ } y, 120 \} \text{ } 154 \cup \\
\text{Fexp} & \text{ } \text{ } \text{ } \{ \text{Cons } (\text{Hd } x) \text{ } (\text{Append } (\text{Tl } x) \text{ } y) \} \text{ } \{ x, 117; \text{ } y, 121 \} \text{ } 152 \\
& \text{=}
\end{align*}
\]

(\text{3 } \times \text{Fexp}, \text{2 } \times \text{Fswt})

\[
\begin{align*}
\text{...}
& \text{(108,109),bind}, \emptyset, 113 \\
\text{Fexp} & \text{ } \text{ } \text{ } [ \text{Null } x ] \text{ } \{ x, 104; \text{ } y, 102 \} \text{ } 108 \cup \\
\text{Fexp} & \text{ } \text{ } \text{ } [ x ] \text{ } \{ x, 104; \text{ } y, 102 \} \text{ } 109 \cup \\
\text{Fexp} & \text{ } \text{ } \text{ } \{ x, 116; \text{ } y, 120 \} \text{ } 154 \cup \\
\text{Fexp} & \text{ } \text{ } \text{ } \{ \text{Cons } (\text{Hd } x) \} \text{ } \{ x, 117; \text{ } y, 121 \} \text{ } 147 \\
\text{Fexp} & \text{ } \text{ } \text{ } \{ (\text{Append } (\text{Tl } x) \text{ } y) \} \text{ } \{ x, 117; \text{ } y, 121 \} \text{ } 148 \\
& \text{=}
\end{align*}
\]

(\text{etc.})

\[\text{Denotational semantics of a flow graph}\]

Figure 7 shows the abstract syntax and the syntactic and semantic domains that are used to describe the semantics of a flow graph. The value `Error` is included as a separate domain to distinguish between certain typing errors and \(\bot\). The semantic function `Fix` is the standard fixed point function. The meaning of a flow graph \(g\) formed from \(n\) function definitions is given by `(\text{Exp } g)` This meaning has the form of a tuple \((\delta_1, \ldots, \delta_n)\) where \(\delta_k \in \text{Dev}, 1 \leq k \leq n\). The function `Def` gives the meaning of an edge in a graph, given a particular environment.

The definitions of the semantic functions `Def` and `Exp` are given in Figure 8. As is standard practice, the domain projection, injection and membership tests have been omitted. The full version of the `Bind` rule would be:

\[
\begin{align*}
\text{Def}(\text{bind }, \emptyset, d; \text{ } g) \text{ } \text{d } \text{ } \text{ } & \text{if } f \in \text{Func} \text{ } \text{then } (f \text{ } \text{| } \text{Func} \text{ } \text{)(Def } \text{ } \text{d } \text{ } \text{b } \text{ } \text{e}) \text{ } \text{else } \bot \text{ } \text{e} \\
& \text{where } f \text{ } \text{= } \text{Def } \text{ } \text{d } \text{ } \text{a } \text{ } \text{e}
\end{align*}
\]

The first argument to the functions `Def` and `Exp` specifies a set of nodes that represent a flow graph, from which the element(s) of current interest are selected by pattern matching.

The `Exp` rule takes all `Export` nodes from the argument set and returns the meaning of the flow graph as a tuple. A given `Export` node with input edge \(a_k\) provides the starting edge for the
Syntactic domains:

\[ a, b, c, d, e \in Edge \quad \text{(Edges are natural numbers)} \]
\[ i \in Ident \quad \text{(Identifiers)} \]
\[ g \in Graph \quad \text{(Power set of vertices)} \]
\[ v \in Value \quad \text{(Constants, such as numbers, True, False and Nil)} \]

Abstract syntax:

\[ G ::= \{V_1 ; \cdots ; V_n\} \quad \text{(A graph is a set of vertices)} \]
\[ V ::= a, \text{LAMBDA}, \emptyset, (d, e) \quad | \quad (a, b), \text{BIND}, \emptyset, d \]
\[ | \quad (a, b), \text{SWITCH}, \emptyset, (d, e) \quad | \quad (a, b, c), \text{MERGE}, \emptyset, d \]
\[ | \quad \emptyset, \text{SOURCE}, v, d \quad | \quad a, \text{SINK}, \emptyset, \emptyset \]
\[ | \quad \emptyset, \text{IMPORT}, i, d \quad | \quad a, \text{EXPORT}, i, \emptyset \]
\[ | \quad a, \text{CALL}, \emptyset, d \quad | \quad a, \text{RETURN}, \emptyset, d \]
\[ | \quad a, \text{USE}, \emptyset, d \]

Semantic domains:

\[ \delta_1, \delta_2, \ldots \in \text{Dev} \equiv \text{Error} + \text{Num} + \text{Bool} + \text{Func} + \{\bot_1; \text{Nil}\} + [\text{Dev} \times \text{Dev}] \]
\[ \epsilon \in \text{Env} = \text{Ident} \rightarrow \text{Dev} \quad \text{(Environments)} \]
\[ \text{Error} = \{\bot_e\} \quad \text{(\bot_e is a type error)} \]
\[ \text{Num} = \{\bot_n; 0; \ldots\} \quad \text{(Domain of numeric values)} \]
\[ \text{Bool} = \{\bot_b; \text{True}; \text{False}\} \]
\[ \text{Func} = \text{Dev} \rightarrow \text{Dev} \]

Semantic functions:

\[ \text{Fix} : \text{Func} \rightarrow \text{Dev} \]
\[ \text{if} : \text{Bool} \rightarrow \text{Dev} \rightarrow \text{Dev} \rightarrow \text{Dev} \]
\[ \text{Def} : \text{Graph} \rightarrow \text{Edge} \rightarrow \text{Env} \rightarrow \text{Dev} \]
\[ \text{Exp} : \text{Graph} \rightarrow \text{Dev} \]

Figure 7: Abstract syntax, and syntactic and semantic domains of a flow graph.
second argument of the application of \(\text{Def}\). The third argument to \(\text{Def}\) is a local environment, which expresses the meaning of all primitive functions and also contains bindings of all \(i_k\) to the fixed point variables \(\delta_k\). This local environment is used by the \(\text{IMPORT}\) clause in the definition of \(\text{Def}\), where \(i\) is either the name of a primitive whose meaning is known, or it is \(i_k\); the name of a function being defined.

Semantic equations:

\[
\begin{align*}
\text{Def}(a, \text{LAMBDA}, \emptyset, (d, e)) & \quad g \; d \; e = \delta_d \\
\text{Def}(a, \text{LAMBDA}, \emptyset, (d, e)) & \quad g \; e = \lambda \delta_d \cdot \text{Def} \; \delta \; a \; e \\
\text{Def}((a, b), \text{BIND}, \emptyset, d) & \quad g \; d \; e = (\text{Def} \; \delta \; a \; e) \; (\text{Def} \; \delta \; b \; e) \\
\text{Def}((a, b), \text{SWITCH}, \emptyset, (d, e)) & \quad g \; d \; e = \text{if} \; \text{Def} \; \delta \; a \; e \; \text{then} \; \text{Def} \; \delta \; b \; e \; \text{else} \; \bot_e \\
\text{Def}((a, b), \text{MERGE}, \emptyset, (d, e)) & \quad g \; e = \text{if} \; \text{Def} \; \delta \; a \; e \; \text{then} \; \bot_e \; \text{else} \; \text{Def} \; \delta \; b \; e \\
\text{Def}(\emptyset, \text{SOURCE}, \text{True}, d) & \quad g \; d \; e = \text{True} \quad \text{(similarly for other constants)} \\
\text{Def}(\emptyset, \text{IMPORT}, i, d) & \quad g \; (i, \delta; e) = \delta \\
\text{Def}(a, \text{CALL}, \emptyset, d) & \quad g \; d \; e = \text{Def} \; \delta \; a \; e \\
\text{Def}(a, \text{RETURN}, \emptyset, d) & \quad g \; d \; e = \text{Def} \; \delta \; a \; e \\
\text{Def}(a, \text{USE}, \emptyset, d) & \quad g \; d \; e = \text{Def} \; \delta \; a \; e \\
\text{Def} & \quad g \; d \; e = \bot_e \quad \text{(iff none of the other \(\text{Def}\) clauses apply)} \\
\text{Exp}(a_1, \text{EXPORT}, i_1, \emptyset; \ldots; a_n, \text{EXPORT}, i_n, \emptyset; g) & = \text{Fix} \; (\lambda \; (\delta_{a_1}, \ldots, \delta_{a_n}) \cdot (\text{Def} \; \delta \; a_1 \; e, \ldots, \text{Def} \; \delta \; a_n \; e)) \\
\text{where} & \quad e = (i_1, \delta_{a_1}, \ldots; i_n, \delta_{a_n}; \text{Primitives}) \\
\text{Primitives} = \{ \text{Cons}, \lambda \delta_1 \cdot \lambda \delta_2 \cdot \text{Cons} \; \delta_1 \; \delta_2 ; \} \\
& \quad \text{Hd} \; , \lambda \delta_1 \cdot \text{Hd} \; \delta_1 ; \\
& \quad \text{Null} \; , \lambda \delta_1 \cdot \text{Null} \; \delta_1 ; \\
& \quad \text{Tl} \; , \lambda \delta_1 \cdot \text{Tl} \; \delta_1 ; \ldots \} \\
\]  

Figure 8: Semantic equations \(\text{Def}\) and \(\text{Exp}\) of a flow graph

The \text{LAMBDA} and \text{SWITCH} nodes are special because they have two outputs rather than one. The second parameter to \(\text{Def}\) decides for which of the two outputs \(d\) or \(e\), the semantics must be given. This is expressed by the non-linear pattern match in the two equations for \text{LAMBDA} as follows:

\[
\begin{align*}
\text{Def}(a, \text{LAMBDA}, \emptyset, (d, e); g) & \quad d \; e = \delta_d \\
\text{Def}(a, \text{LAMBDA}, \emptyset, (d, e); g) & \quad e = \lambda \delta_d \cdot \text{Def} \; \delta \; a \; e \\
\text{Def} & \quad g \; x \; e = \bot_e \\
\equiv & \quad \text{Def}(a, \text{LAMBDA}, \emptyset, (d, e); g) \; x \; e = \text{if} \; x = d \; \text{then} \; \delta_d \\
& \quad \text{else if} \; x = e \; \text{then} \; \lambda \delta_d \cdot \text{Def} \; \delta \; a \; e \\
& \quad \text{else} \; \bot_e \\
\]  

The first \text{LAMBDA} clause states that the name of the bound variable corresponding to the \text{LAMBDA} node is to be a unique identifier, here generated by \(\delta_{\text{edge number}}\) (all edge numbers are unique).
The meaning of the other output edge of the \texttt{LAMBD A} node as given by the second \texttt{LAMBD A} clause is a $\lambda$-abstraction that binds the variable introduced by the first \texttt{LAMBD A} clause. In the semantics of \texttt{Append} to be presented below, the variables $\delta_{102}$ and $\delta_{104}$ are introduced by the \texttt{LAMBD A} clauses.

The meaning of both the $d$ and $e$ outputs of the \texttt{SWITCH} node depends on the meaning of the control input $a$ and that of the value input $b$. The first \texttt{SWITCH} clause states that if the meaning of the control input is \texttt{True}, the meaning of the $d$ output is the same as that of the $b$ input. If the meaning of the control input is \texttt{False} that of the $d$ output is irrelevant, which is indicated by $\perp_e$. This is to ensure that the meaning of all edges is defined, whether actually used or not. The control input $a$ of a \texttt{SWITCH} node is intended to also steer a \texttt{MERGE} node (via a network of \texttt{USE} nodes), so that both may operate in synchrony. In a correct program, a $\perp_e$ value derived from a \texttt{SWITCH} node never passes through a \texttt{MERGE}.

The derivation of the semantics for the flow graph of \texttt{Append} from Figure 4 is shown below. At each step only the node is shown that is of current interest, i.e. the one whose output edge matches the current value of the second argument to \texttt{Def}. Each invocation of \texttt{Def} has the entire set of nodes at its disposal. The first five steps of the derivation are:

\[
\begin{align*}
\text{Exp } g &= \text{Exp (105, EXPORT, Append, 0);} & g_1 \\
&= \text{Fix (} \lambda \delta_{105} \cdot \text{Def (103, LAMBD A, 0, (104, 105)}; & g_2 105 \epsilon) \\
&= \text{Fix (} \lambda \delta_{105} \cdot (\lambda\delta_{104} \cdot \text{Def (101, LAMBD A, 0, (102, 103)}; & g_3 103 \epsilon) \\
&= \text{Fix (} \lambda \delta_{105} \cdot (\lambda\delta_{104} \cdot (\lambda\delta_{102} \cdot \text{Def (159, RETURN, 0, 101}; & g_4 101 \epsilon) \\
&= \text{Fix (} \lambda \delta_{105} \cdot (\lambda\delta_{104} \cdot (\lambda\delta_{102} \cdot \text{Def ((156, 154, 152), MERGE, 0, 159}; & g_5 159 \epsilon) \\
\ldots \\
\text{where} \\
\epsilon &= (\text{Append, } \delta_{105}; \text{Primitives})
\end{align*}
\]

The process continues for a while until a fixed point expression is reached. We found the process so tedious that we wrote a small functional program as suggested by Stoy [22] to do the rewriting. The resulting fixed point expression for \texttt{Append} is:

\[
\begin{align*}
\text{Exp } g &= \text{Fix (} \lambda \delta_{105} \cdot (\lambda\delta_{104} \cdot (\lambda\delta_{102} \cdot \text{if Null } \delta_{104} \text{ then } \delta_{102} \\
&\text{ else Cons (Hd } \delta_{104} (\delta_{105} (\text{TL } \delta_{104} \delta_{102} ))))) &
\end{align*}
\]

This concludes the presentation of the syntax and semantics of the flow graph that is at the heart of the compiler.

\section*{PROGRAM ANALYSIS ON THE FLOW GRAPH}

The compiler performs a variety of major program analyses, such as strictness, boxing and sharing analysis. In addition a number of minor analyses are performed to support the major analyses. The compiler synthesizes code based on the findings of the analysis phases up to the point where a complete C program has been generated.

Each analysis and synthesis function is formally specified as a set of rewrite rules on a flow graph set, together with a set of attributes that represent the information gathered by the analysis. A rewrite rule on a flow graph set selects nodes and attributes, and replaces the selected items by a new set of items. In most cases, the rewrite rules only add one or more attributes to the flow graph set, thus gradually increasing the information about the program being compiled. In the compiler, the analysis and synthesis functions are applied in a certain order, so that each pass adds a further set of attributes to the flow graph set. At the end of the compilation process, the C code generated for the functional program is present as a set of attributes, which only need to be printed so as to form the compiled program.
All major analyses performed by the compiler (i.e., not just strictness analysis but also boxing analysis etc.) are based on Wadler's linear domain [23]. The principles of the analyses will be described using the simplest domain possible that still provides useful information. This two-point domain consists of the points 0 and 1. The meaning of these values is as follows: 0 signifies that "absolutely no information" is present and 1 means that "all information" is present. It depends on the analysis what "information" actually means. In the context of strictness analysis the point 1 means that a head normal form is required. In the context of boxing analysis it means that an object is required that need not be stored in the heap.

In the following sections a backwards and a forwards analysis are described over the two-point domain. The extension of the analysis over a larger domain is discussed in some detail in the section "Projection analysis".

Strictness analysis: a backwards analysis

The principle of strictness analysis is the observation, that at run time, a function application is evaluated only if the answer is really needed. At compile time strictness analysis derives the information required to implement this as follows: the "seed" for the analysis process is planted at the exit (i.e., the \texttt{RETURN} node) of a function. By obeying the rewrite rules of the analysis, the strictness information travels upwards to the entries of a function (i.e., the \texttt{LAMBDA}, \texttt{SOURCE} and \texttt{IMPORT} nodes). Restricting our attention to the two-point domain allows for the mechanism to be discussed in detail, without having to describe at the same time methods to find fixed points (see section "Recursive function definitions and fixed point iteration"). The strictness analysis is given by the function $U$ in Figure 9.

For each type of node in the flow graph, there is a rule in an analysis or synthesis function. Such a function either propagates information downwards, from the entry points towards the exits of a flow graph, or upwards, from the exit points towards the entry points. The $U$ function propagates information upwards. The rewrite rules that perform upwards propagation take attributes for the output edges of each node and compute the attributes for the input edges. Downwards propagation reverses the overall flow of information so it requires the rewrite rules to take attributes on their input edges and to produce attributes for the output edges. The $U$ analysis defines an upwards flow, whereas the $D$ function, also in Figure 9 defines a downwards flow. Often a complete analysis or synthesis phase requires a combination of an upwards and a downwards flow, which thus requires two separate functions, and two separate attributes.

The use of a flow graph thus requires that the direction of the information flow is made explicit in the program analysis and synthesis functions. This contrasts with the way an attribute grammar propagates information. The flow of information implicit in the computation of inherited attributes in an attribute grammar corresponds to upwards propagation in a flow graph. The information flow in calculating synthesised attributes in an attribute grammar corresponds to downwards information propagation in a flow graph. Since the flow graph formalism enforces the use of separate analysis and synthesis functions for the production of separate attributes, it can be considered as a specification method that is of a lower level than an attribute grammar. On the other hand, a flow graph is simpler to implement in an efficient way.

In the next two sections, the analysis functions given in Figure 9 will be applied to the flow graph for \texttt{Append} to explain exactly how analysis on the flow graph takes place.

\textit{The first argument of \texttt{Append}}

The $U$ analysis determines the strictness of the arguments of a function represented as a flow graph. The strictness is represented by \texttt{SUP} attributes (for Strictness UP). As the $U$ function performs backwards analysis, the nodes are visited from bottom to top, starting with the \texttt{RETURN} node. The analysis is "seeded" at the input edges \texttt{a} of the \texttt{RETURN}, \texttt{SINK} and \texttt{EXPORT} nodes.
\[ \text{Figure 6: Strictness and availability analysis functions } D \text{ and } D_{x} \]
Figure 9 shows that all these nodes are allowed to “fire” without requiring already existing \texttt{SUP} attributes; all other rewrite rules need an existing \texttt{SUP} attribute to fire. In the \textit{Append} example, the first \texttt{SUP} attribute that appears in the flow graph set is generated by the \texttt{return} on edge \texttt{59}. The attribute value \texttt{1} states that a head normal form is associated with edge \texttt{59}. The relevant subset of the flow graph set that represents \textit{Append} and its attributes is shown below. Attributes are shown next to the edge to which they belong. The arrows on the edges indicate the direction of the information propagation:

\[
(156, 154, 152), \texttt{merge}, \emptyset, 159 \\
\uparrow \texttt{sup}, 159, 1 \\
159, \texttt{return}, \emptyset, 101
\]

To help understand the method, the reader may wish to annotate the edges in Figure 4 with the value of the \texttt{SUP} attributes as the derivation proceeds.

The second rewrite step taken by the \texttt{U} function is to combine the newly generated attribute on edge \texttt{59} with the \texttt{merge} node that connects to edge \texttt{59}. The \texttt{merge} rule in Figure 9 specifies that the formal attribute value \texttt{ud} (actual value \texttt{1}) as received on the formal \texttt{d}-edge (actual value \texttt{59}) of the \texttt{merge} node is to be sent on formal edges \texttt{a}, \texttt{b} and \texttt{c}. In the \textit{Append} example this means that the attribute value \texttt{1} now appears on edges \texttt{56}, \texttt{54} and \texttt{52} respectively. Only one of these attributes, that for edge \texttt{56}, is shown below, together with the relevant part of the \textit{Append} flow graph set:

\[
113, \texttt{use}, \emptyset, 156 \\
\uparrow \texttt{sup}, 156, 1 \\
(156, 154, 152), \texttt{merge}, \emptyset, 159 \\
\uparrow \texttt{sup}, 159, 1 \\
159, \texttt{return}, \emptyset, 101
\]

Now three rewrites are possible: one involving the \texttt{use} node on edge \texttt{56}, one involving the \texttt{use} node on edge \texttt{54} and one involving the \texttt{call} node on edge \texttt{52}. The choice here is to proceed with the first possibility, so the next rewrite rule to consider is the \texttt{use} rule in Figure 9. This rule states that to compute the value of the \texttt{SUP} attribute on the \texttt{a}-edge, the \texttt{SUP} attributes for all \texttt{use} nodes that have the same \texttt{a}-edge must be combined using the logical or (\texttt{\lor}). In the \textit{Append} example, this means that \texttt{SUP} attributes on edges \texttt{56}, \texttt{115}, and \texttt{119} are in principle required first, before progress can be made. However, the \texttt{\lor} operator has the nice algebraic property that if one of its operands is \texttt{1}, the value of the remaining operands does not matter, for the answer will always be \texttt{1}. The \texttt{SUP} attribute on edge \texttt{56}, which has just been calculated by the \texttt{merge} rule, is \texttt{1}, so the \texttt{use} rule may now set the \texttt{SUP} attribute on edge \texttt{113} to \texttt{1}, regardless of the values of the \texttt{SUP} attributes on edges \texttt{115} and \texttt{119}. This yields the following (partial) state of the flow graph set:

\[
110, \texttt{call}, \emptyset, 113 \\
\uparrow \texttt{sup}, 113, 1 \\
113, \texttt{use}, \emptyset, 156 \\
\uparrow \texttt{sup}, 156, 1 \\
(156, 154, 152), \texttt{merge}, \emptyset, 159 \\
\uparrow \texttt{sup}, 159, 1 \\
159, \texttt{return}, \emptyset, 101
\]

The fourth rewrite is an application of the \texttt{call} rule in Figure 9, which copies the \texttt{SUP} attribute.
The fifth rewrite rule to be applied is the \texttt{BIND} rule. Figure 9 shows that the \texttt{BIND} rule not only requires a \texttt{SUP} attribute on the \texttt{d}-edge, but also the \texttt{ABS} attribute on the \texttt{b}-edge, as well as the \texttt{SUP} attribute on some other edge \texttt{e} as specified by \texttt{ABS}, \texttt{b}, \texttt{e}. The \texttt{ABS} attributes are generated by the \texttt{A} scheme, which is not described in the paper because it is similar to the \texttt{ld} scheme in the way attributes are propagated. The \texttt{A} scheme works out connections between formal and actual arguments of functions. A formal argument is identified with the \texttt{d}-edge of a \texttt{LAMBDA} node, whereas an actual argument is identified with the \texttt{b}-edge of a \texttt{BIND} node. In the \texttt{Append} example, the \texttt{ABS} attribute value on edge 137 is edge 104, to note the correspondence between an actual first argument (137) of \texttt{Append} and the formal first argument (104). Similarly, the \texttt{ABS} attribute value carried by edge 142 is edge 102.

Before continuing the derivation two problems should be considered: the first is how to deal with higher order functions and the second is how to represent predefined properties of primitive functions. Both problems are effectively solved by the one and the same mechanism: add unconnected edges to the flow graph.

When the compiler does not know a function by its name, the \texttt{ABS} attribute points at an unconnected edge that carries a set of attributes with “no information”. This means that absolutely no information is available about functions passed as a parameter, or functions returned by other functions (see also section “Projection analysis”).

When the compiler needs information about a primitive function, the \texttt{ABS} attribute points at an unconnected edge that carries all the required information. For example, the \texttt{ABS} attribute on edge 108 points at the edge corresponding to the formal argument edge (say edge \texttt{Null}) of \texttt{Null}. One of the required properties, the \texttt{SUP} attribute, specifies whether \texttt{Null} is a strict function. This is indeed the case: to determine whether the end of a list has been reached, the top level constructor of the list must be evaluated to head normal form. The attribute specifying this fact is thus \texttt{SUP,\texttt{Null},1}.

Adding primitive functions to the compiler thus means allocating one or more extra edges and entering the desired properties for these edges in a table. This makes it easy to extend and alter the set of primitives supported by the compiler.

Returning to the \texttt{BIND} rule in Figure 9 we see that it can now be applied, since \texttt{Null} is a primitive and all the attributes specified in the \texttt{BIND} rule are present. The computation involved in the generation of the new attribute on the \texttt{b}-edge (edge 108 in case of the \texttt{Append} example) requires taking the logical and ($\land$) of the \texttt{SUP} attribute on the \texttt{e} edge (\texttt{Null}) and the incoming \texttt{SUP} attribute on the \texttt{d}-edge (110). The value of both \texttt{SUP} attributes is 1, so the new value of the \texttt{SUP} attribute on edge 108 is 1. In the flow graph set below, the three relevant attributes are shown next to edge 108. Only the first is a newly generated attribute, the \texttt{ABS} attribute is generated by a previous compilation scheme and the attribute \texttt{SUP,\texttt{Null},1} is loaded into the

\begin{verbatim}
(109,108),BIND,\emptyset,110
    \textsuperscript{\texttt{SUP,110,1}}
110,CALL,\emptyset,113
    \textsuperscript{\texttt{SUP,113,1}}
113,USE,\emptyset,156
    \textsuperscript{\texttt{SUP,156,1}}
(156,154,152),MERGE,\emptyset,159
    \textsuperscript{\texttt{SUP,159,1}}
159,RETURN,\emptyset,101
\end{verbatim}
The fifth rewrite brings a SUP attribute to the formal argument edge (104) of the Append's first argument. The attribute value is 1, which means that Append is strict in its first argument:

104, USE, 0, 108
   \uparrow SUP, 108, 1; ABS, 108, Null_1; SUP, Null_1, 1
   (109,108), BIND, 0, 110
      \uparrow SUP, 110, 1
   110, CALL, 0, 113
      \uparrow SUP, 113, 1
   113, USE, 0, 156
      \uparrow SUP, 156, 1
   (156,154,152), MERGE, 0, 159
      \uparrow SUP, 159, 1
   159, RETURN, 0, 101

This rather lengthy discussion is really a proof of the fact that, given the rules of Figure 9, if a head normal form is required as output of Append, there must be a head normal form on the first argument of Append. In the usual formal notation we have just proved that:

\forall x,y : (Append x y \neq \bot) \quad \Rightarrow \quad (x \neq \bot)  
\quad \Rightarrow \quad (\text{contra positive rule})

The rewrite rules of Figure 9 are the deduction rules used in the strictness proof.

The second argument of Append

The second argument to Append is not a strict argument, but how will the proof system deal with it? Consider again the MERGE rule in Figure 9. The MERGE rule states that the SUP attribute on the d-edge is copied verbatim onto the a-, b- and c-edges. Recalling the meaning of the SUP attribute, the MERGE rule thus says that if a head normal form is required on the d-edge, then normal forms are also required on the b- and c-edges. These last two edges represent the expressions in the then and else branches of a conditional, so in effect the MERGE rule suggests that the conditional is strict in all its arguments! This is not the case, as the MERGE rule is simply a formalisation of the way code will eventually be generated for the conditional in the C output. The use of the C "if { } else { }" construct in the generated code (c.f. Figure 3)
guarantees that only one of the two branches will be evaluated. Which one it is depends on
the boolean expression. So the boolean expression must always be evaluated, and it is safe
to generate code to also evaluate the then- and else-branches of the conditional. When the
generated C program is actually executed, only the branch chosen by the boolean expression
will actually be evaluated and the code for the other branch will be skipped. The part of the flow
graph set that corresponds to tracing the flow graph through the else-branch of the conditional
is shown below:

\[
\begin{align*}
&149, \text{CALL}, \emptyset, 152 \\
&\uparrow \text{SUP}, 152, 1 \\
&(156, 154, 152), \text{MERGE}, \emptyset, 159 \\
&\quad \uparrow \text{SUP}, 159, 1 \\
&159, \text{RETURN}, \emptyset, 101
\end{align*}
\]

The call rule copies the sup attribute on its d edge (152) over to its a edge (149):

\[
\begin{align*}
&(148, 147), \text{BIND}, \emptyset, 149 \\
&\quad \uparrow \text{SUP}, 149, 1 \\
&149, \text{CALL}, \emptyset, 152 \\
&\quad \uparrow \text{SUP}, 152, 1 \\
&(156, 154, 152), \text{MERGE}, \emptyset, 159 \\
&\quad \uparrow \text{SUP}, 159, 1 \\
&159, \text{RETURN}, \emptyset, 101
\end{align*}
\]

The next rewrite to consider is another bind application, this time involving the second argument
of the primitive function Cons. This function is not strict in either of its arguments, so the formal
argument edges for Cons (edge Cons1 for the first argument and Cons2 for the second) both
carry a sup attribute value 0. With this information the sup attribute calculated by the bind
rule yields 0 for the sup attribute on edge 147 (because \(1 \wedge 0 = 0\)):

\[
\begin{align*}
&144, \text{CALL}, \emptyset, 147 \\
&\quad \uparrow \text{SUP}, 147, 0; \text{ABS}, 147, \text{Cons2}; \text{SUP}, \text{Cons2}, 0 \\
&(148, 147), \text{BIND}, \emptyset, 149 \\
&\quad \uparrow \text{SUP}, 149, 1 \\
&149, \text{CALL}, \emptyset, 152 \\
&\quad \uparrow \text{SUP}, 152, 1 \\
&(156, 154, 152), \text{MERGE}, \emptyset, 159 \\
&\quad \uparrow \text{SUP}, 159, 1 \\
&159, \text{RETURN}, \emptyset, 101
\end{align*}
\]

It is not difficult but a bit tedious to continue the reasoning up to the point where the switch
rule has been reached, because once the sup attribute has turned to 0, no call or bind rule
can deliver the value 1. A single use node such as that on edge 120, or on edge 121 copies the
sup attribute on the d-edge over to the a-edge. Such a “single” use node cannot alter a sup
attribute. Hence the state of the relevant part of the flow graph set after the switch rule has
fired is as shown below. Here the sup attributes for both edges 120 and 121 are shown:

\[
\begin{align*}
&102, \text{CALL}, \emptyset, 118 \\
&\quad \uparrow \text{SUP}, 118, 0 \\
&(119, 118), \text{SWITCH}, \emptyset, (120, 121) \\
&\quad \text{SUP}, 120, 1 \uparrow \quad \text{SUP}, 121, 0
\end{align*}
\]
The final rewrite step brings the `SUP` attribute home to the `LAMBD A` node corresponding to the second argument of `Append`:

\[
\begin{align*}
101, \text{LAMBD A}, \emptyset, (102, 103) \\
&\uparrow \text{SUP}, 102, 0 \\
102, \text{CALL}, \emptyset, 118 \\
&\uparrow \text{SUP}, 118, 0 \\
(119, 118), \text{SWITCH}, \emptyset, (120, 121) \\
\text{SUP}, 120, 1 &\uparrow \uparrow \text{SUP}, 121, 0
\end{align*}
\]

This second derivation shows that with the rules as given by \( \mathcal{U} \) in Figure 9, it is not possible to prove that `Append` is strict also in its second argument. Hence the compiler must assume that it is non-strict for safety reasons.

**Availability analysis: a forwards analysis**

The compiler uses the strictness information present in the form of `SUP` attributes to decide when head normal forms are present at run time as follows: an expression on an edge that carries a `SUP` attribute value of 1 is in head normal form. An expression on an edge that carries a `SUP` attribute of 0 may, or may not be in head normal form. This way of using strictness analysis is not sufficient for the compiler to generate good code. This can be explained as follows. The part of the `Append` flow graph that represents applications of `Hd` and `Tl` contains only `SUP` attribute values of 0 (the edges involved are 122–124, 127, 132–134 and 137). Because of this, the code generator must assume that the actual arguments presented to `Hd` (edge 122) and `Tl` (edge 132) are not head normal forms, for a value 0 represents “no information”. This is a safe, but overly pessimistic assumption because in actual fact the expressions associated with edges 122 and 132 are head normal forms. The problem is that the necessary information is present in the flow graph, but not at the right place. The information is present at edge 104, which carries a `SUP` attribute of 1 because it represents a strict argument. What needs to be done now is to transfer this information explicitly to edges 122 and 132. This is one of the purposes of the \( \mathcal{D} \) function in Figure 9.

We shall now trace the execution of the \( \mathcal{D} \) function up to the point where all the information necessary to generate the code for `Append` as shown in Figure 3 is available. The first rewrite rule of interest is the `LAMBD A` rule, which “reflects” the `SUP` attribute on the \( b \)-edge as a `SDN` attribute on the same edge. The edges now point downwards to make the direction of the information flow explicit:

\[
\begin{align*}
103, \text{LAMBD A}, \emptyset, (104, 105) \\
&\downarrow \text{SUP}, 104, 1; \text{SDN}, 104, 1 \\
104, \text{USE}, \emptyset, 114
\end{align*}
\]

The `USE` rule propagates the `SDN` attribute on its input edge to its output edge yielding:

\[
\begin{align*}
103, \text{LAMBD A}, \emptyset, (104, 105) \\
&\downarrow \text{SUP}, 104, 1; \text{SDN}, 104, 1 \\
104, \text{USE}, \emptyset, 114 \\
&\downarrow \text{SDN}, 114, 1 \\
(115, 114), \text{SWITCH}, \emptyset, (116, 117)
\end{align*}
\]

The `SWITCH` rule also propagates the `SUP` attribute on its data input edge to both its output
edges. Only one of the generated attributes is shown below:

\[
\begin{array}{l}
103, \text{LAMBDA}, \emptyset, (104, 105) \\
\downarrow \text{SUP}, 104, 1; \text{SDN}, 104, 1 \\
104, \text{USE}, \emptyset, 114 \\
\downarrow \text{SDN}, 114, 1 \\
(115, 114), \text{SWITCH}, \emptyset, (116, 117) \\
\downarrow \text{SDN}, 117, 1 \\
117, \text{USE}, \emptyset, 122 \\
\end{array}
\]

The last step of interest is another application of the \text{USE} rule, which delivers the \text{SDN} attribute to the \text{BIND} node.

\[
\begin{array}{l}
103, \text{LAMBDA}, \emptyset, (104, 105) \\
\downarrow \text{SUP}, 104, 1; \text{SDN}, 104, 1 \\
104, \text{USE}, \emptyset, 114 \\
\downarrow \text{SDN}, 114, 1 \\
(115, 114), \text{SWITCH}, \emptyset, (116, 117) \\
\downarrow \text{SDN}, 117, 1 \\
117, \text{USE}, \emptyset, 122 \\
\end{array}
\]

The propagation of the \text{SDN} attribute from edge 104 to edge 132 proceeds in a similar way. The \text{SDN} attributes for the remaining edges are also propagated, but these are not essential. The code generation function now has sufficient information to generate good code, as it “knows” that head normal forms will always be associated with edges 108, 123 and 132. This particular information permits the use of the macros \text{Null}, \text{Hd} and \text{Tl} in the C code for \text{Append}:

\[
\text{return Cons}(\text{Hd}(x), \text{Suspend}(\text{PrelAppend}, \text{Tl}(x), y));
\]

Without the information that \(x\) is a head normal form, the generated code would have had to construct suspensions for the \text{Hd} and \text{Tl} functions as follows:

\[
\text{return Cons}(\text{Suspend}(\text{PrelHd}, x), \text{Suspend}(\text{PrelAppend}, \text{Suspend}(\text{PrelTl}, x), y));
\]

The unoptimised version of the code is considerably worse, because of the relatively heavy cost of implementing the suspensions.

**Dealing with conditionals in a non-strict context**

The \(D\) function serves a second purpose. At the introduction of the use of the C “\{ \} else \{ \}” construct in the compiled code, it has been tacitly assumed that the corresponding conditional expression in the functional program appears in a strict context. This is indeed the case in the \text{Append} function. However, a conditional might also appear in a non-strict context, for example as one of the arguments to \text{Cons} thus:

\[
\text{Foo} x\ y\ z = \text{Cons} (\text{if } x \text{ then } y \text{ else } z) \text{ Nil}
\]

In this case, the C code should build a suspension of the conditional function rather than try to execute the conditional. The desired code is thus:

\[
\text{return Cons}(\text{Hd}(x), \text{Suspend}(\text{PrelAppend}, \text{Tl}(x), y));
\]
To achieve this goal, there are two switch rules and two merge rules in the $D$ function. If the conditional expression appears in a non-strict context (the sdn attribute on the control inputs then has the value 0), the first switch rule matches. This rule replaces the switch node by use and sink nodes. The first merge rule then also matches, which replaces the merge node by bind and call nodes. After these two changes have been made, the conditional expression is represented by an ordinary chain of bind nodes, just like any other function application. The allocation mechanism of unique edge labels takes this possibility into account. This is one of the reasons why the edge labels in Figure 4 are not consecutive numbers.

If the conditional expression appears in a strict context (the sdn attribute on the control inputs has the value 1), the second merge rule and the second switch rule match. This does not change the structure of the flow graph, but simply propagates the sdn attribute.

In either case, the attributes that are necessary for further processing must be added explicitly to the flow graph set. This requires some care on behalf of the designer of flow graph rewrite rules, because it is quite easy to miss an attribute out, or attach an attribute to the wrong edge. In general, a flow graph is not so well suited to applications that make many changes to the structure of the graph. For this reason, our compiler consists of two major parts: the first part implements the passes that make large changes to the program represented as an ordinary parse tree. The second part performs mainly program analysis and synthesis on the flow graph set, while only making a few small changes to the structure.

**Recursive function definitions and fixed point iteration**

The derivation of the strictness properties of $Append$ as they have been presented in the previous three sections are incomplete in the sense that the presence of a recursive call to $Append$ has not been taken into account. In the particular case of $Append$ this issue could safely be ignored, as the recursive call does not affect the strictness or non-strictness of the arguments. In general the recursive calls cannot be ignored. The denotational semantics make this point explicit by using the fixed point combinator $\text{Fix}$ (see Figure 8) to derive the real meaning of the flow graph for $Append$. As program analysis derives an abstract meaning of functions, there must also be an element of fixed point computation. Finding fixed points during program analysis is a rather difficult subject to which a significant body literature is devoted [1].

For the two-point domain fixed points can be found as follows: Define an initial approximation to the strictness of each formal argument of all functions in a compilation unit. This approximation must necessarily assume that no user defined function is strict in any of its arguments. As a consequence, the sup attributes for the formal argument edges of a user defined function start with the value 0. By preloading these attributes into the flow graph set, as well as the attributes for the primitive functions, as has been assumed earlier, all formal argument edges will carry a sup attribute. Fixed point iteration can simply consist of running the analysis function ($\mathcal{U}$) a number of times, until the sup attribute values have reached a stable value (the fixed point). It is prudent to limit the number of repetitions allowed, although in practice we found that not even the larger benchmark programs used require more than six iterations. In fact very little improvement to the approximation is gained beyond the second iteration. This method thus implements the fixed point iteration at the level of the flow graph analysis.

If a more detailed domain than the two-point domain is used this method should not be used for the following reason. In the two-point domain, an attribute either carries no information (0)
or all information (1). This allows for only one transition to be made (from 0 to 1). Once an attribute reaches the value 1, it can never change again. Enlarging the number of points in the domain will also enlarge the number of possible transitions, so it will take longer for attributes to reach a stable value. Therefore, when more detail is required than the two-point domain has to offer, the fixed point iteration must be built into the representation of the attributes. This means, that instead of working with values, such as 0 and 1, the attribute values must represent dependencies of the form:

\[
\text{Append is strict in its second argument if Append is strict in its second argument}
\]

\[
\text{and if Cons is strict in its second argument.}
\]

Since Cons is non-strict in any of its arguments, the only conclusion that can be drawn from this statement is that Append is not strict in its second argument. However, when considering more detailed information, the situation changes considerably. The next section will show that the flow graph framework also supports analysis where detailed, contextual information plays an important role.

**Projection analysis**

Projection analysis extends strictness analysis to reasoning about higher order functions (i.e. functions that take functions as arguments or functions that return other functions) and/or structured data (as opposed to atomic data) [23, 24, 25, 26, 27, 28]. Common to all approaches is the search for a non-flat domain that allows useful information to be determined at a reasonable cost. The most ambitious approach uses an infinite domain of arbitrary structure. The two-point domain [29] thus far described is the simplest domain that will provide useful information. The information gathered by strictness analysis of a particular function is always valid; that is it is independent of the context in which the function is used. Projection analysis gathers more detailed information, which takes into account in which context a function is used. This information is thus only valid in the appropriate context.

Projection analysis allows a compiler to generate code that is specifically optimised towards the intended context. Consider as an example a call to Append in a context where the entire spine of the result list is known to be needed. A spine is a chain of list constructor nodes, linked via their tail fields and ending at a Nil node. The head elements of the constructor nodes may assume any value. A spine is shown below as a list \([x_1, \ldots, x_n]\) of arbitrary values. When asking Append to produce a spine, it is safe to evaluate both its arguments to spines before calling Append. This is safe because even if the computation for one of the arguments to a spine diverges, the call to Append will also diverge. Schematically we have the result of strictness analysis and that of projection analysis with respect to a spine as:

\[
\begin{align*}
\text{Append} & \Downarrow y \\
\text{Append} & \{x_1, \ldots, x_n\} \{y_1, \ldots, y_m\} = \{x_1, \ldots, x_n, y_1, \ldots, y_m\} (\text{Strictness analysis})
\end{align*}
\]

A spine is just one example of a useful context with respect to lists. There are many other contexts that are also useful for lists and even more contexts for arbitrary data structures.

The next two sections provide a brief description of the implementation of projection analysis in the flow graph compiler.

**Structured data**

The flow graph compiler performs analysis over Wadler’s linear domain [23], as shown in Figure 10. This allows the compiler to infer certain properties of functions that operate on lists. The domain is not detailed enough to gather information about arbitrary list structure or any other data structures.
We found it more convenient to represent points in the linear domain by numbers in the range 0 to $N$ rather than a string of $\infty$, $\bot$ and $\in$ symbols. ($N$ is specified as a compiler option). This numerical notation has been borrowed from Burn [24].

<table>
<thead>
<tr>
<th>Wadler</th>
<th>Burn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty \in \mathbb{N}$</td>
<td>$2n + 1$ — where $n = N \div 2 \land N$ is odd</td>
</tr>
<tr>
<td>$\bot \in \mathbb{N}$</td>
<td>$2n$ — where $n = N \div 2 \land N$ is even</td>
</tr>
<tr>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td>$\bot \in \in$</td>
<td>$4$ — a spine of spine normal forms</td>
</tr>
<tr>
<td>$\in \in$</td>
<td>$3$ — a spine of head normal forms</td>
</tr>
<tr>
<td>$\bot \in \in$</td>
<td>$2$ — a spine normal form</td>
</tr>
<tr>
<td>$\in$</td>
<td>$1$ — a head normal form</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$0$ — no information</td>
</tr>
</tbody>
</table>

Figure 10: The correspondence between Wadler’s and Burn’s domain nomenclature

<table>
<thead>
<tr>
<th>function/argument</th>
<th>PRO attributes ($N$ is a compiler option)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Null}$</td>
<td>PRO, $\text{Null}_1$, $[0 \ 1 \ 1 \ 1 \ 1 \ldots \ 1]$</td>
</tr>
<tr>
<td>$\text{Cons/first}$</td>
<td>PRO, $\text{Cons}_1$, $[0 \ 0 \ 0 \ 1 \ 2 \ldots \ N - 2]$</td>
</tr>
<tr>
<td>$\text{Cons/second}$</td>
<td>PRO, $\text{Cons}_2$, $[0 \ 0 \ 2 \ 3 \ 4 \ldots \ N]$</td>
</tr>
<tr>
<td>$\text{Hd}$</td>
<td>PRO, $\text{Hd}_1$, $[0 \ 1 \ 1 \ 1 \ 1 \ldots \ 1]$</td>
</tr>
<tr>
<td>$\text{Tl}$</td>
<td>PRO, $\text{Tl}_1$, $[0 \ 1 \ 2 \ 2 \ 2 \ldots \ 2]$</td>
</tr>
</tbody>
</table>

Figure 11: The primitives used in $\text{Append}$ and their projection attributes

A projection attribute, as delivered and used by the projection analysis in the compiler, is a vector: \text{PRO}, \text{a}, \lfloor n_0 \ldots \ n_N \rfloor, \text{where} \ n_k \in \{0 \ldots \ N\}. The elements of a projection vector say that given a function that is called in an \text{i}-style context, the expression in the function body that is associated with edge \text{a} may be evaluated using the $n_i$-style. An element of the projection attribute can thus be viewed as an evaluation transformer [25, 26]. The projection attributes for the primitive functions used in $\text{Append}$ are shown in the second column of Figure 11. Vector elements are counted starting from 0. Element 1 gives the minimum requirements that must be satisfied for the function to produce a 1-style (head normal form) result, element 2 gives the requirements for a 2-style (spine normal form) result etc. For example $\text{Cons}$ has a 0 at position 1 in the projection attribute for both its arguments, so it requires no more than 0-style (unreduced) inputs to produce a 1-style result. The remaining primitives shown are strict and thus require at least a 1-style argument to produce a 1-style result.

The next vector elements (at position 2, which corresponds to a spine strict context) show a greater variety. The $\text{Null}$ test function needs to inspect the top level list constructor only, to decide whether a list is empty or not; it never needs access to the tail of any list, hence the 1 at positions $1 \ldots N$ of its projection vector. The $\text{Hd}$ function ignores the tail of a list constructor so here again a 1-style argument is required. The second argument to $\text{Cons}$ and the argument to $\text{Tl}$ must both be 2-style if a 2-style result is needed.

At the next level up in the abstract domain (position 3) $\text{Tl}$ cannot insist on a 3-style argument, as this would imply evaluation to head normal form of the head of the list, which might not be needed. It is safe however for $\text{Cons}$ to require a 3-style second argument. A 1-style first argument is now in order too, because it will become the new head of a 3-style list.

The information in Figure 11 can now be used as the starting point for projection analysis. We will not discuss the analysis here, but refer the reader to a technical report [30]. Instead we
will show the results of the analysis.

The code that is generated by the compiler for functions to be called in the various context styles, implements compile time choice [31, 32] of versions. This means that for each possible context (1, 2, 3 etc.) the compiler creates a version of those functions that are capable of delivering the right style of list. For Append there is version that produces a head normal form but also a version that delivers a spine. The C code generated for Append that may be used in a context that requires a spine, is:

```c
ptr SpineAppend(x, y)
ptr x, y;
{
    if (Null(x)) {
        return y;
    } else {
        return Cons(Hd(x), SpineAppend(Tl(x), y));
    }
}
```

Comparing SpineAppend to Append (see Figure 3) shows that the former is considerably more efficient. The first difference is, that SpineAppend does not build a suspension for the recursive call. This is safe, because upon entry, both arguments x and y to SpineAppend are already evaluated to spines. Therefore Tl(x) will also be a spine, so the invariant that the arguments must both be spines is satisfied for the recursive call. Because of the call-by-value semantics of C programs, Cons evaluates its arguments, so when a call to SpineAppend returns, the entire result will have been created to form a new spine. The second difference between the two C versions compiled for Append is that there is no call to the reducer in the then-branch of SpineAppend. Such a call is unnecessary because the argument y is guaranteed to be a spine when SpineAppend is entered.

The compiler organisation does not depend on the fact that Wadler’s linear domain is used, although that makes it quite straightforward to build and solve the fixed point equations. Changing the domain would require a rework of several of the analyses. It would be interesting to investigate what changes have to be made and how the quality of the generated code would be affected by using an abstract domain with a finer structure.

**Higher order functions**

The flow graph compiler does not perform any of the analysis taking higher order functions into account. In spite of this apparent shortcoming, the compiler generates good code even for applications that use higher order functions. The compiler generates a specialised copy of a function that takes another function as an argument. Consider the definition of the function Map and a main expression below. The first actual argument of Map is the trigonometric function Sin:

1. Map : (→ *) → [→] → [→]
2. Map \( f \) \( x = \) if \( \text{Null} \; x \) then Nil
   else Cons (f (Hd x)) (Map f (Tl x))
3. 4. Main = Map Sin (Cons 2 Nil)

Specialisation transforms the program above into the equivalent program below. The task of the higher order function Map is taken over by the first order function Map', which has the Sin
function built in. Because of this, the \( \text{Map}' \) function is not a polymorphic function:

1. \( \text{Map}' : [\text{Num}] \rightarrow \)
2. \( \text{Map}' \quad x = \text{if} \quad \text{Null} \quad x \quad \text{then} \quad \text{Nil} \)
3. \( \quad \text{else} \quad \text{Cons} \quad (\text{Sin} \quad (\text{Hd} \quad x)) \quad (\text{Map}' \quad (\text{Tl} \quad x)) \)
4. \( \text{Main} = \text{Map}' \quad (\text{Cons} \quad 2 \quad \text{Nil}) \)

Specialisation thus lessens the need for analysis over higher order functions, although it does not remove all higher order functions.

IMPLEMENTING A FLOW GRAPH ANALYSIS FUNCTION IN C

A flow graph and the associated forwards and backwards functions are at the same time powerful enough to express useful analysis and synthesis functions and simple enough to enable an efficient implementation. The purpose of this section is to present a simplified version of the implementation in C of the \( \text{ul} \) function from Figure 9. The required data structures are introduced first, followed by the implementation of the analysis function.

Data structures representing nodes, edges and attributes

The nodes and attributes in a flow graph are implemented as a C structure. The \( a, b \) and \( c \) fields in the node structure are the input edges and the \( d \) and \( e \) fields are the output edges. The attributes appear as fields of the \( \text{attrib} \) structure. The \( \text{abs} \) and \( \text{sup} \) fields record the \( \text{ABS} \) and \( \text{SUP} \) attributes. The purpose of the \( \text{fst} \) field will be explained shortly.

```c
typedef struct {
    int a,b,c;
    int type;
    int qualifier;
    int d,e;
} node;

typedef struct {
    int abs,sup,fst;
} attrib;
```

The nodes are accessed via two arrays of pointers to the nodes. The \( \text{up} \) array is used by the backwards analyses, and the \( \text{dn} \) is used by the forwards analyses. The node pointers in these two arrays are ordered such that attributes are always present when they are required. The \( \text{up} \) and \( \text{dn} \) arrays are traversed from beginning to end (see below). The variable \( \text{node\_count} \) holds the number of nodes that are present in the flow graph. The \( \text{node\_count} \) variable and the \( \text{up} \) and \( \text{dn} \) arrays are setup when the flow graph is built out of a parse tree. Unless nodes are added or removed (which is rare), these data structures remain unchanged. The cost of building the sorted arrays is thus amortised over many analysis and synthesis passes.

```c
#define MAX_NODES 200000
node * up[MAX_NODES];
node * dn[MAX_NODES];
int node_count;
```

Attributes are stored in a separate array \( \text{edge} \). Space is reserved to store all attributes of interest on each possible edge.
The 11 node types are defined as constants and some access macros are also defined:

```c
#define BIND 1
#define CALL 2
#define EXPORT 3
#define IMPORT 4
#define LAMBDA 5
#define MERGE 6
#define RETURN 7
#define SINK 8
#define SOURCE 9
#define SWITCH 10
#define USE 11
```

```c
#define TYPE(i) (up[i]->type)
#define SUP(x) (edge[x].sup)
#define ABS(x) (edge[x].abs)
#define FST(x) (edge[x].fst)
```

**The C implementation of the \( \mathcal{U} \) function**

The code of the C version of the \( \mathcal{U} \) scheme is given below. It consists of two parts: the first part initialises the \( \text{SUP} \) attributes on the edges that correspond to the formal arguments of all user defined functions, so that the implementation of the \( \text{BIND} \) rule (see Figure 9) will always find a \( \text{SUP} \) attribute.

```c
void U() {
    int i, a, b, c, d, e, x;
    for (i = 0; i < node_count; i++) {
        d = up[i]->d;
        if (TYPE(i) == LAMBDA) {
            SUP(d) = 0;
        }
    }
}
```

The second part of the function \( \mathcal{U} \) visits each node of the flow graph, starting at the \( \text{RETURN} \) nodes and propagating the \( \text{SUP} \) attribute values towards the \( \text{LAMBDA} \) nodes. The for loop below corresponds to a single pass through the flow graph. To implement fixed point iteration, the entire for loop should be performed a number of times, as discussed in the section “Recursive function definitions and fixed point iteration”.

```c
for (i = 0; i < node_count; i++) {
    a = up[i]->a; b = up[i]->b; c = up[i]->c;
    d = up[i]->d; e = up[i]->e;
    switch (TYPE(i)) {
        case LAMBDA:
            SUP(a) = 0;
            break;
        case BIND:
            x = ABS(b);
            SUP(a) = SUP(d);
            SUP(b) = SUP(d) && SUP(x);
    }
}
```
break;
case SWITCH:
    SUP(a) = 0;
    SUP(b) = SUP(d) && SUP(e);
    break;
case MERGE:
    SUP(a) = SUP(b) = SUP(c) = SUP(d);
    break;
case SOURCE:
    break;
case IMPORT:
    break;
case SINK:
    SUP(a) = 0;
    break;
case EXPORT:
    SUP(a) = 0;
    break;
case CALL:
    SUP(a) = SUP(d);
    break;
case RETURN:
    SUP(a) = 1;
    break;
case USE:
    if (FST(d)) {
        SUP(a) = SUP(d);
    } else {
        SUP(a) |= SUP(d);
    }
    break;
}

As indicated in the code, there is a slight complication in treatment of the use node. The conditional in the USE case yields True only when the first USE node with a particular input edge is encountered. In this case the output attribute (SUP(d)) is copied over to the input edge. Any subsequent output attributes are "or-ed" with the first in the else clause.

There is a close correspondence between the \( U \) scheme and the C routine \( U \). For each rule in the \( U \) scheme there is a corresponding case in the \( U \) routine. The actions in a rule, which consist of adding an attribute value, correspond to assigning the attribute value to a field of the appropriate edge. Because such a simple correspondence exists, it is not difficult to implement a compilation scheme without special program manipulation tools.

The control structure required to implement a set of rewrite rules over a flow graph is a single for loop, that visits the nodes in the right order. The time complexity of the implementation is thus \( O(\text{node count}) \). By allocating all the required data structures at once, the space complexity of an analysis routine is constant.

RESULTS

To assess the effectiveness of the method, the results of the analyses is compared with related work and some performance results for both the compiler and the generated code are presented.
Figure 12 presents a few list functions taken from papers by Wadler [23] and Nöcker [33].

The infix operator : has been used instead of the Cons function to improve readability. The analysis results in the 4-point domain \{0;1;2;3\} are shown for each argument as projection vectors (see also Figure 11).

Wadler’s method gives the same results as ours. The method used by Nöcker gives better results because he uses a domain with a finer structure, which allows information about the head of a list to be represented. In Wadler’s linear domain this is not possible.

Wadler:

\[
\begin{align*}
Rev x &= \text{if } \text{Null } x \text{ then } Nil \text{ else } \text{Append } (Rev (Tl x)) \ (Hd x : Nil) \\
Last s &= \text{Hd } (\text{Rev } s) \\
Append x y &= \text{if } \text{Null } x \text{ then } y \text{ else } Hd x : \text{Append } (Tl x) \ y
\end{align*}
\]

Nöcker:

\[
\begin{align*}
\text{Foldr } op \ r x &= \text{if } \text{Null } x \text{ then } r \text{ else } op \ (Hd x) \ (\text{Foldr } op \ r \ (Tl x)) \\
\text{Catenate } l &= \text{Foldr } \text{Append } Nil \ l \\
\text{Concat } a \ b \ c &= \text{Catenate } (a : b : c : Nil)
\end{align*}
\]

<table>
<thead>
<tr>
<th>function</th>
<th>Nöcker</th>
<th>Wadler/Flow graph compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Argument</td>
<td></td>
</tr>
<tr>
<td>Rev</td>
<td>[0 \ 2 \ 2 \ 3]</td>
<td>[0 \ 2 \ 2 \ 3]</td>
</tr>
<tr>
<td>Last</td>
<td>[0 \ 2 \ 2 \ 2]</td>
<td>[0 \ 2 \ 2 \ 2]</td>
</tr>
<tr>
<td>Append</td>
<td>[0 \ 1 \ 2 \ 3]</td>
<td>[0 \ 1 \ 2 \ 3]</td>
</tr>
<tr>
<td>Foldr</td>
<td>[0 \ 0 \ 0 \ 0]</td>
<td>[0 \ 0 \ 0 \ 0]</td>
</tr>
<tr>
<td>Concat</td>
<td>[0 \ 1 \ 2 \ 3]</td>
<td>[0 \ 0 \ 2 \ 3]</td>
</tr>
<tr>
<td>Catenate</td>
<td>[0 \ 1 \ 3 \ 3]</td>
<td>[0 \ 1 \ 3 \ 3]</td>
</tr>
</tbody>
</table>

Figure 12: Some list processing functions and the projection attributes of the arguments

Compiler performance

The first two compiler passes build a flow graph from programs in Intermediate syntax. All remaining passes generate, manipulate and use attributes associated with the edges of the flow graph. Table 1 lists the individual passes applied to the lambda program, one of the benchmarks that we have available. The source text, without blank lines and comments, comprises 770 lines, 276 functions. Of the total 54.22 seconds compilation time (on a SUN SPARC station 1 with 16 MB memory) more than 45% is spent on the code generation (passes 32–36). The strictness analysis (passes 16 and 17) takes less than 5%. We will not go into the detail of the remaining passes but briefly mention some of the more interesting ones. Cheap eagerness causes expressions to be evaluated in a lazy context when it is definitely cheaper to do so than it would be to build a suspension [34, 31]. The sets of edges that carry normal forms in some style determined by strictness/projection analysis and cheap eagerness analysis are combined in pass 20. Passes 29–31 assist the compiler in flattening the flow graph when printing the generated code.

As described in the previous section, the flow graph compiler visits the nodes of the flow graph in a particular order. The nodes have to be sorted when new nodes appear or when nodes are made redundant. This implies that due to the first sort phase the time complexity of the compiler will be at least \(O(n)\), where \(n\) represents the number of nodes in the flow graph. This number fluctuates somewhat, but stays within 20% of the average (see the nodes column in Table 1). Subsequent sorts may be implemented with much better performance, or removed altogether by the use of a more sophisticated data structure.
Depending on the size $N + 1$ of the abstract domain and the particular analysis there may be some expensive operations that have to be performed for each node. It is not possible to give an a priori upper bound for the complexity of these operations.

**Runtime performance**

Some experiments have been performed with the compiler to assess the performance of the method. The benchmark programs are the result of collecting performance data on implementations of functional programming languages [35, 36]. The programs used are:

1. *em script* runs a simple script through a functional implementation of the Unix text editor [37];
2. *gcode* compiles a simple program into scalar G-machine code according to the compilation schemes as described by Johnsson [34];
3. *lambda* $(S K K)$ evaluates to $I$ on an implementation of the $\lambda$-K calculus [36];
4. *sched $7$* calculates an optimum schedule of $7$ parallel jobs with a branch and bound algorithm [38];
5. *wave* $3$ predicts the tides in a rectangular estuary of the North Sea over a period of $3 \times 20$ minutes [38].

A more detailed description of the run-time organisation than we have described here may be found in our paper [31], where we also argue that the statistic that provides the best indication for the quality of our method is the number of cell claims made. Table 2 shows the cell claims (as reported in our other papers [35, 36]) witnessed by Turner's standard combinator reduction machine [18], an early version of Johnsson's G-machine [34] 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 it is safe to 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 2 show a significant decrease in the number of claimed cells when moving from standard combinators via the G-machine to the flow graph compiler. This is as expected because standard combinators are virtually unoptimised; the G-machine compiler (as used to obtain the figures reported here) performs strictness analysis but only locally within a function and the flow graph compiler carries various forms of analyses across function boundaries.

The set of cell claim figures labelled 2-point $(N = 1)$ apply to a version of the compiler and runtime system, where the abstract domain is restricted to two points [3]. The second set of cell claim figures show that extending the abstract domain to three points $(N = 2)$ gives a reduction in cell claims of about 25% for the bigger programs, which includes the advantage of using vector application nodes instead of chains of binary application cells.

The *wave* program does not perform so well as the other programs on the G-machine, because it essentially requires fully-lazy lambda lifting, which the G-machine compiler does not provide. The flow graph compiler does not perform full-lazy lambda lifting either, but 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
Compiler pass | nodes | sec.
--- | --- | ---
1 parse input | 0 | 1.26
2 build the flow graph | 15853 | 3.85
3 sort nodes | 15853 | 1.20
4 connect SINK nodes to dangling edges | 17234 | 0.21
5 sort nodes | 17234 | 1.36
6 constant folding | 17417 | 0.25
7 sort nodes | 17356 | 1.45
8 remove redundant nodes | 17891 | 0.13
9 sort nodes | 13006 | 1.04
10 propagate function contexts | 13006 | 0.15
11 generate ABS attributes and CALL nodes | 14822 | 0.18
12 sort nodes | 14822 | 1.14
13 evaluate constant expressions | 14909 | 0.10
14 sort nodes | 14909 | 1.27
15 generate WID attributes | 14909 | 0.13
16 build equations for strictness analysis | 14909 | 1.81
17 solve equations | 14909 | 0.70
18 build equations for cheap eagerness analysis | 14909 | 1.16
19 solve equations | 14909 | 0.61
20 combine strictness and cheap eagerness | 14990 | 1.07
21 sort nodes | 14969 | 1.24
22 remove redundant nodes | 15494 | 0.11
23 sort nodes | 14925 | 1.43
24 group USE nodes with equal input edges | 14925 | 0.11
25 build equations for boxing analyses | 14925 | 3.32
26 solve boxing equations | 14925 | 1.98
27 build equations for sharing analysis | 14925 | 1.17
28 solve sharing equations | 14925 | 0.66
29 flag tail calls | 14925 | 0.06
30 derive total ordering on the nodes | 14925 | 0.21
31 compute compile time reference counts | 14925 | 0.08
32 generate expressions | 14925 | 4.91
33 gather lists of temporaries | 14925 | 0.72
34 generate declarations | 14925 | 5.34
35 remove redundant versions of functions | 14925 | 0.20
36 generate function bodies | 14925 | 13.61

Total seconds | 54.22

Table 1: Passes in the flow graph compiler with the size of the flow graph (in nodes) and the execution time of each pass (in seconds as measured on a SPARC Station 1)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>em</th>
<th>gcode</th>
<th>lambda</th>
<th>sched</th>
<th>wave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinators</td>
<td>733742</td>
<td>966963</td>
<td>77128</td>
<td>477243</td>
<td>424623</td>
</tr>
<tr>
<td>G-machine</td>
<td>245008</td>
<td>144709</td>
<td>18058</td>
<td>42246</td>
<td>403645</td>
</tr>
<tr>
<td>Flow graph (2-point)</td>
<td>125101</td>
<td>57571</td>
<td>9110</td>
<td>25485</td>
<td>16078</td>
</tr>
<tr>
<td>Flow graph (3-point)</td>
<td>85625</td>
<td>40954</td>
<td>7018</td>
<td>19591</td>
<td>11953</td>
</tr>
</tbody>
</table>

Table 2: Cell claims for the benchmark programs with various compilers
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.

We have extensively benchmarked the compiler and compared it to compilers for other lazy functional languages (LML [15], Haskell [8] and Clean [39]). The results show that the code generated by our compiler is often faster than that generated by other compilers [40].

CONCLUSIONS

An important task of an optimising compiler for a functional language is to minimise the number of cells claimed by the compiled code. Several techniques are available that help to reduce cell claims, most notably strictness analysis. A framework has been presented that allows the available techniques to be integrated in an efficient fashion.

The program being compiled is represented as a flow graph, which is basically an abstract syntax tree with some extra nodes and edges. Their presence allows both forwards and backwards abstract interpretation to be performed with equal ease. This was found to be essential as strictness analysis is inherently more efficient when performed by backwards rather than forwards abstract interpretation [41]. Other analyses are most naturally performed in a forwards manner.

Most information processed by forwards and backwards abstract interpretations flows along the edges in a flow graph. Sometimes information has to travel along a path that is not explicitly traced. In such a case an auxiliary analysis is defined to establish the required paths, thus separating the issues of how to get the information where it is required and what the information itself should be.

A flow graph is a lean structure since the only primitive that has been built into it (apart from function application and abstraction) is the conditional. Other primitive functions can be added or deleted without changing the structure of a flow graph. The properties of primitive functions are defined by a number of tables, which are easily modified and extended.

An algorithm has been presented to compile a functional program into a flow graph and to give the denotational semantics of a flow graph. This provides the theoretical underpinning of the flow graph framework.

The compiler makes a large number of passes over the flow graph, adding, using and removing information piecemeal. The compiler has fine control over both the time and space complexity of the compilation process. The space and time complexity of the analyses in terms of the number of attributes processed are essentially $O(n)$ where $n$ represents the number of nodes in the flow graph that represent the program being compiled.

The space requirements of the compiler are further reduced because of its organisation in many different passes. This makes it possible to decide when information will not be needed any more, such that the storage this information occupies can be reclaimed. For instance, after incorporating information carried by the SUP attributes into the SDN attributes, all storage occupied by the SUP attributes can be reclaimed. In the compiler separate attributes are stored in separate lists of linked blocks, so that such reclaims are cheap and easy.

We have thus described the structure of a working system that can efficiently compile lazy functional languages with a high level of analysis and a sound theoretical foundation.

ACKNOWLEDGEMENTS

We thank Marcel Beemster, Andy Gravell, Chris Hankin, Koen Langendoen, Theodosis Papathanassiadis, Wim Vree and the referees for their comments on draft versions of the paper. Stuart Bell has helped us to implement the compiler.
This work was supported by the Science and Engineering Research Council, UK, under grant No. GR/F 35081, FAST: Functional programming for arrays of Transputers.

References


