URL: http://www.elsevier.nl/locate/entcs/volume89.html 22 pages

Redux: A Dynamic Dataflow Tracer

Nicholas Nethercote[1] and Alan Mycroft[2]

Computer Laboratory, University of Cambridge
Cambridge, United Kingdom

Abstract

Redux is a tool that generates *dynamic dataflow graphs*. It generates these graphs by tracing a program's execution and recording every value-producing operation that takes place, building up a complete computational history of every value produced. For that execution, by considering the parts of the graph reachable from system call inputs, we can choose to see only the dataflow that affects the outside world. Redux works with program binaries, and thus is not restricted to programs written in any particular language.

We explain how Redux works, and show how dynamic dataflow graphs give the *essence* of a program's computation. We show how Redux can be used for debugging and program slicing, and consider a range of other possible uses.

1 Introduction

Redux is a tool that generates *dynamic dataflow graphs* (DDFGs). These graphs represent the entire computational history of a program.

1.1 Overview

Redux supervises a program as it executes, and records the dataflow—inputs and outputs—of every operation that produces a value. Every register and word of memory is shadowed by a pointer to a sub-graph that shows how the value was computed.

A program's behaviour, as seen from the outside world, is entirely dictated by the system calls it makes during execution (assuming we ignore timing issues). This includes its exit status, which comes from the `_exit()` system call. Once a program terminates, by considering the parts of the DDFG reachable from system call inputs, we can show all the computations that

[1] Email: `njn25@cam.ac.uk`
[2] Email: `am@cl.cam.ac.uk`

directly affect this behaviour.[3] This captures the *essence* of the program's computation, and ignores uninteresting book-keeping details.

Redux runs on x86 machines running Linux, works directly with program binaries without requiring any recompilation, relinking, or even source code, and is language-independent.

1.2 Contributions of this Work

The contributions of this work are as follows.

- We introduce the dynamic dataflow graph. It is basically the same as Agrawal and Horgan's *dynamic dependence graph* (DDG) [4] minus the control nodes and edges, at a lower level of abstraction (instructions instead of C statements), and with the arrows reversed. However, we claim that ignoring control flow is crucial for making the graphs more widely useful.
- We introduce a tool, Redux, which computes DDFGs. Redux is the first tool we know of that traces program computations in detail without being tied to a particular programming language or language implementation.
- We show that these graphs are not just data structures that need to be built to achieve another goal, such as program slicing, but useful in their own right for program visualisation and other uses, and that improving their presentation is important.
- We claim that DDFGs represent the *essence* of a program's computation, and show that programs that compute the same thing in highly different ways have very similar or identical DDFGs.

1.3 Paper Organisation

This paper is structured as follows. Section 2 introduces DDFGs with some examples. Section 3 describes how Redux works. Section 4 shows how programs computing the same thing in very different ways can have similar or identical DDFGs. Section 5 discusses possible uses of Redux. Section 6 describes difficulties and problems with Redux. Section 7 discusses related work, and Section 8 concludes.

2 Dynamic Dataflow Graphs

A dynamic dataflow graph (DDFG) is a directed acyclic graph (N, E). N is a multi-set of nodes representing value-producing operations (with multiple executions of the same operation represented distinctly). E is the set of edges denoting dynamic dataflow between nodes; it also contains some edges denoting state dependencies that are important to capture the program's behaviour. This section gives some example DDFGs produced by Redux.

[3] We will use "behaviour" in this sense throughout this paper.

```
int faci(int n) {
   int i, ans = 1;
   for (i = n; i > 1; i--)
      ans = ans * i;
   return ans;
}

int main(void) {
   return faci(5);
}
```

```
int facr(int n) {
   if (n <= 1)
      return 1;
   else
      return n * facr(n-1);
}

int main(void) {
   return facr(5);
}
```

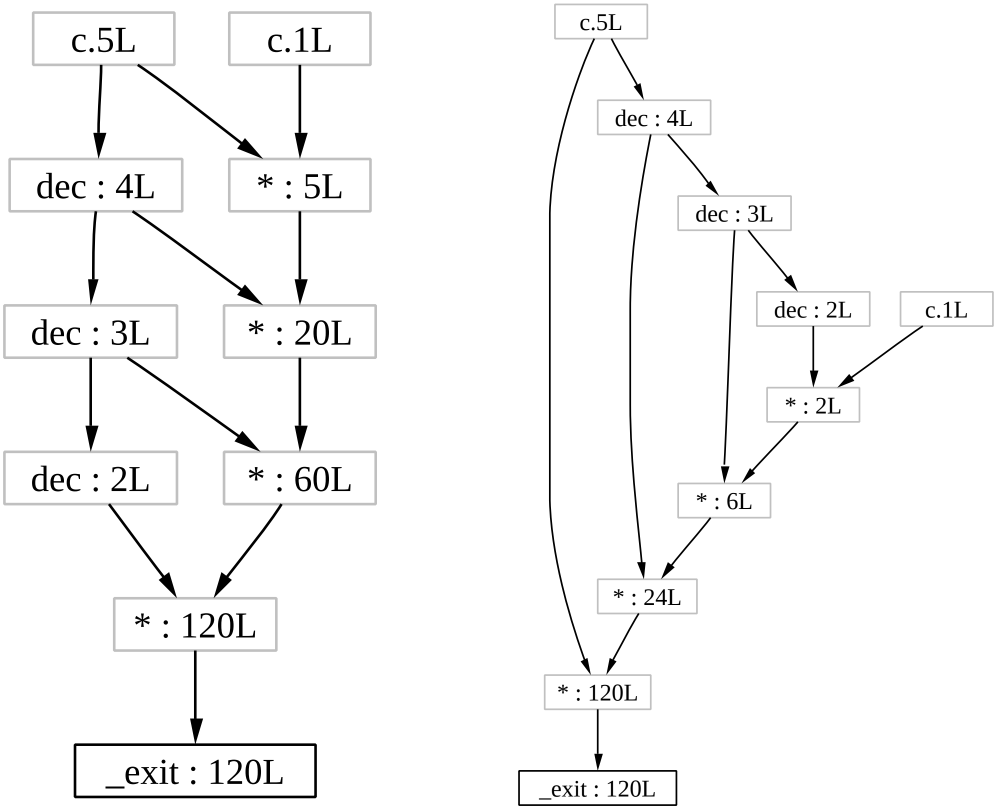


Fig. 1. Iterative and recursive `fac(5)` in C

2.1 Factorial

Figure 1 shows two C programs that compute the factorial of five, one iteratively, and one recursively, and the resultant DDFGs. There are three node types shown. Those labelled “c.5L” and “c.1L” represent constants. The ‘c’ (short for “code”) indicates that the constants appeared literally in the code; the letter “L” (short for “long”) indicates they have 32 bits. Operation nodes (e.g. “dec : 4L”; “dec” is short for “decrement”) contain an operation name colon-separated from the operation’s result; the operands can be seen from the input edges. The “_exit” node shows the program’s exit status, as returned by the `_exit()` system call; system call nodes are shown with darker borders.

All operations that produce values are represented with nodes—mostly arithmetic/logic operations, and system calls. Other book-keeping computations that do not produce values are not shown, such as loads from and stores to memory (pointer use), register-to-register moves, and function calls (direct and indirect). Also, no control flow is shown in the DDFG (it will be considered further in Section 6.3). Module boundaries are also irrelevant.

The DDFG represents values produced, and the inputs to the operations producing them. It does not represent the exact location of those inputs and outputs, the address of the instructions that produced them, and so on.

2.2 Hello World

Figure 2 shows the Hello World C program and two versions of its DDFG. Consider first the larger DDFG on the left-hand side. We will first explain the DDFG's components, and then analyse it, to see what it tells us about the program's execution. We will then consider the smaller DDFG.

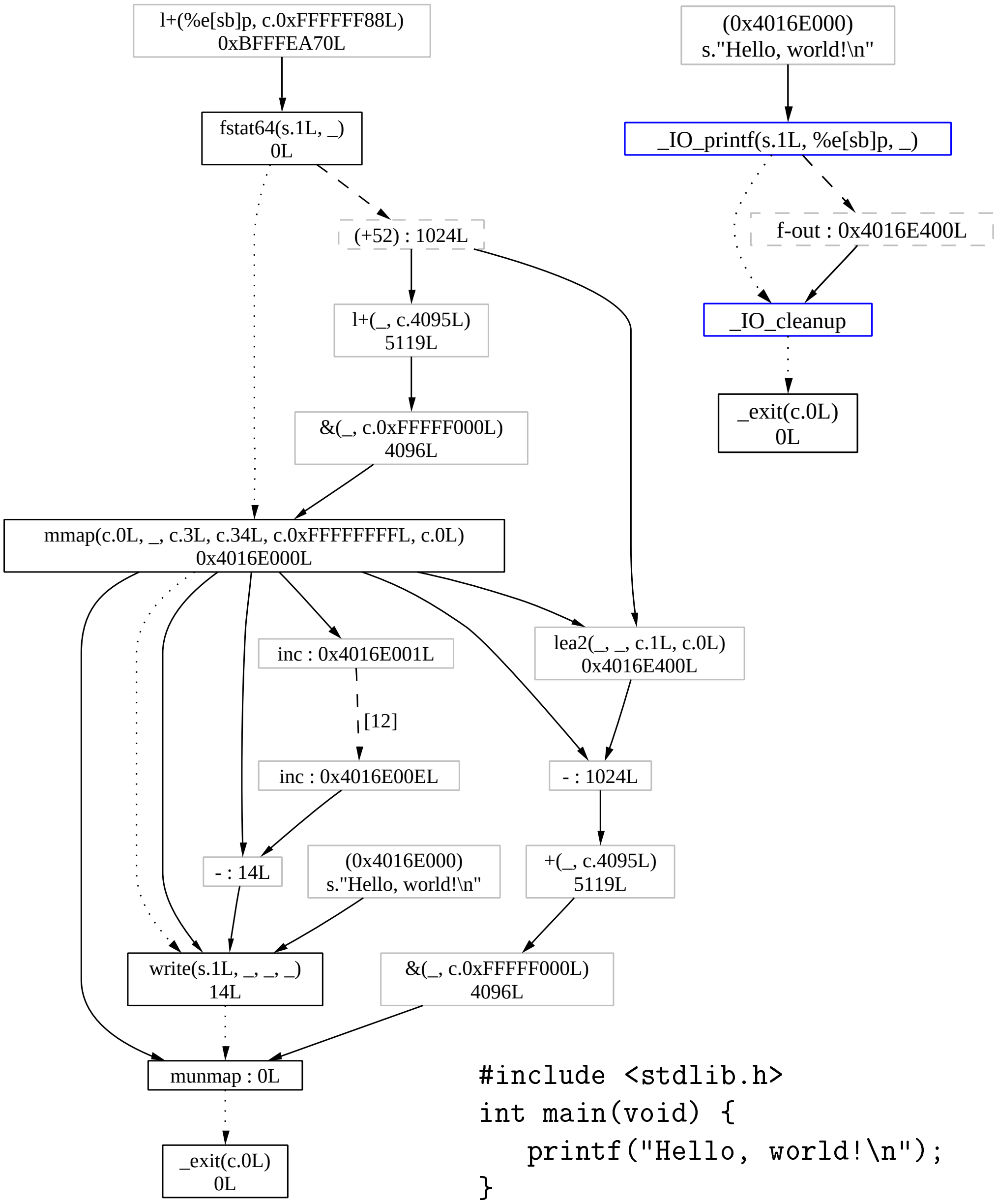
Static constants are written with an "s." prefix. The system call temporal order, which must be preserved because they have side-effects affecting the program's behaviour, is shown by the dotted state dependency edges. The sequencing of other operations is unimportant, except that the obvious dataflow constraints must be observed.

System call arguments are shown in the same way as arithmetic operations' operands. However, many system calls also have indirect arguments, accessed via direct pointer arguments; these *memory inputs* are important too. In this example, the "Hello, world!\n" memory input is shown as an extra argument to the `write()` system call. Its address is given in parentheses to show that it is paired with the direct pointer argument. System calls can also produce *memory outputs* via pointers. They are shown with dashed edges to dashed nodes, which contain the offset of the word within the memory block.[4] The `fstat64()` call has one; this is the `.st_blocksize` field 52 bytes into the output `struct stat64` buffer.

The top node contains an inlined argument "%e[sb]p". By default, the computational histories of the stack pointer (`%esp`) and frame pointer (`%ebp`) are not tracked. This is because the computations producing these pointers are usually long, uninteresting chains of small constant additions and subtractions. The "l+" nodes represent `lea` x86 instructions, which programs often use for integer addition.

Even for this tiny program, several measures have been taken to make the DDFG as compact and readable as possible: non-shared constants are inlined into their descendent nodes; a chain of increment ("inc") nodes is abbreviated, with a dashed edge indicating the number of elided nodes; and the string argument to `write()` is pretty-printed in a compact way.

[4] This makes sense because Linux system calls only use pointers to indirectly access flat memory blocks; there are no reads within linked data structures.



```
#include <stdlib.h>
int main(void) {
   printf("Hello, world!\n");
}
```

Fig. 2. Hello World

Now for the analysis. We can interpret the program's operation quite precisely from the DDFG. The function `printf()` checks that standard output (file descriptor 1) is writable with the `fstat64()` system call. It then allocates a buffer using `mmap()` (at 0x4016E000). The static string "Hello, world!\n" is copied into the buffer, which is passed to the `write()`. The other arguments to `write()` are 1 (standard output again) and the string length (14) which is the difference between the string's start and end addresses; the end address was found by incrementing a variable for each character copied into the buffer (shown by the abbreviated chain of "inc" nodes). Finally, the buffer is freed with `munmap()`, and `_exit()` terminates the program.

Finally, Redux can also create more abstract DDFGs, by using its `--fold` option to specify functions whose nodes and edges should be conflated. The right-hand side of Figure 2 shows Hello World's DDFG when `_IO_printf()` and `_IO_cleanup()` are folded.[5] The function `_IO_cleanup()` is invoked by the C library after `main()` exits; we had to look through the `glibc` source code to determine its name. Function output ("f-out") nodes are similar to system call memory output nodes—they show values produced by the function (unlike system call nodes, this includes its return value) that are subsequently used in a computation reachable from a system call argument.

3 How Redux Works

3.1 Implementation Framework

Redux is implemented as a *skin* that plugs into Valgrind [13], a generic program supervision framework. Valgrind executes programs using dynamic binary translation, and thus can instrument all code in the program, including libraries, without needing source code. Valgrind handles the complicated task of executing the supervised program. Redux only has to instrument the code, and provide runtime support for the instrumentation.

3.2 Overview

Graphs are built dependence-wise, i.e. each node points to its inputs. This is the natural way to do things, since an operation's operands are created before the operation takes place, and each operation has a fixed number of operands (except for system calls with memory inputs) whereas an operation result can be used any number of times. However, the graphs are drawn dataflow-wise, i.e. each node points to the nodes that use its value. We have found this more intuitive to read.

The basic implementation idea is simple. Every register and memory word is *shadowed* by a pointer to a graph node, showing how it was computed. The graph is initially empty. To begin, register shadows are initialised to point to a special "unknown register value" node, and each word of static memory has its shadow initialised (see Section 3.7 for details). As the program progresses, one node is added to the graph for each value-producing instruction executed. At termination, Redux draws only those nodes in the graph that directly affected the system call arguments, and hence the program's behaviour.

It is worth noting that the instrumentation added does not change the basic behaviour of a program; the instrumentation's effects take place "beside" the program's execution. In particular, the instrumentation does not affect whether a program terminates.

[5] We folded `_IO_printf()` because that is the name that appears in `glibc`'s symbol table, which Valgrind relies on to detect entry to the function; `printf()` is just an alias.

3.3 Building and Sharing Nodes

Nodes hold a tag indicating their type ("+", "inc", etc.) and pointers to their operands. During execution, a new node is built for every value-producing operation executed. Let us denote the shadow pointer for register `%reg` with `sh(%reg)`. This instruction (using AT&T assembly code syntax):

```
addl %eax, %ebx
```

will be instrumented to update the shadow registers as follows:

```
sh(%ebx) := +(sh(%ebx), sh(%eax))
```

Where `+(X,Y)` represents a "+" node. Instructions that only move existing values, such as moves, loads, and stores, are instrumented so that the shadows are copied appropriately. For example:

```
movl %ebx, %ecx
```

is instrumented to update the shadow registers as follows:

```
sh(%ecx) := sh(%ebx)
```

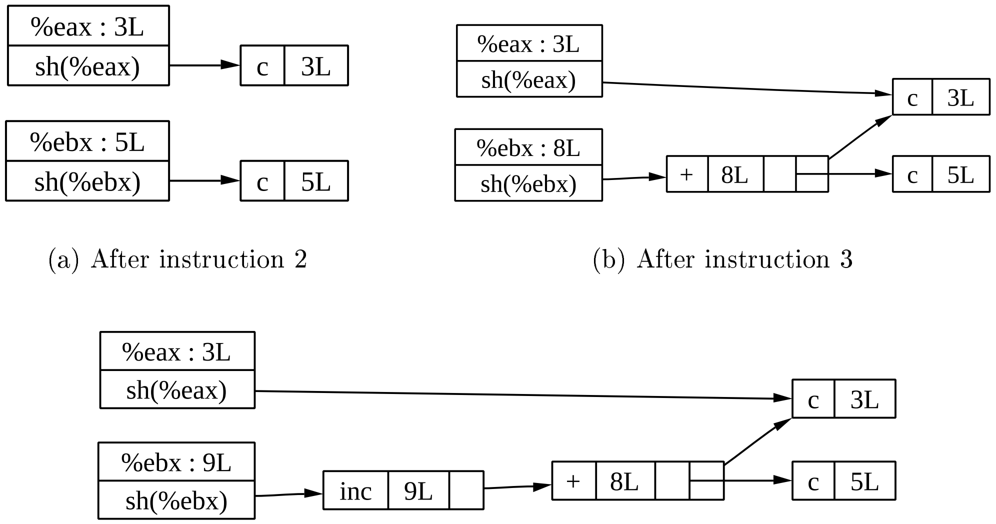
Consider the following instructions:

```
1: movl $3,   %eax
2: movl $5,   %ebx
3: addl %eax, %ebx
4: incl %ebx
```

For instructions 1 and 2, two constant nodes are built; `sh(%eax)` and `sh(%ebx)` are set to point to them, as shown in Figure 3(a). For instruction 3, a "+" node is built, with the two constant nodes as its operands, and `sh(%ebx)` is updated to point to it (Figure 3(b)). Each node stores the result of its operation, and the "+" node's result (8) is equal to the value in `%ebx`. This is an invariant: the result of a node pointed to *directly* by a register or memory word's shadow is always equal to the value of the register or memory word. For instruction 4, an "inc" node is built, its operand being the "+" node, and `sh(%ebx)` is updated again (Figure 3(c)). Again the invariant holds—`%ebx`'s value (9) matches the "inc" node's, but no longer matches the "+" node's, which `sh(%ebx)` now only indirectly points to.[6]

The shadows of deallocated memory words are set to point to a special "unknown memory" node. If these nodes appear in a program's graph, it very probably indicates a bug in the program.

[6] Valgrind uses a RISC-like intermediate representation called UCode that is much simpler than x86 code. Skins do not see x86 code directly, so these examples are not quite representative. But the ideas and results are the same.



(a) After instruction 2

(b) After instruction 3

(c) After instruction 4

Fig. 3. Building a Graph

3.4 System Calls

System calls are not handled very differently to arithmetic operations. When one takes place, the arguments (and their shadows) can be found from the registers (and shadow registers) easily.

Memory inputs are trickier, but not too difficult because Valgrind provides hooks that tell a skin when a system call reads memory. When this happens, Redux gathers the shadow words for the memory input block into an aggregate "chunk" node which is made an additional input to the system call node.

Memory outputs are handled similarly. When Valgrind tells Redux that a system call has written memory, Redux builds new system call memory output nodes for each word written by the call.

3.5 Sub-word Operations

Register and memory shadowing is done at the word (32-bit) level. However, some instructions use one- or two-byte operands. This requires special "read byte" and "read word" nodes for extracting sub-words, and "split" nodes for combining them. Consider the instruction that moves the least-significant byte of register `%eax` into `%ebx`:

```
movb %al, %bl
```

It is instrumented to update the shadow registers as follows:

```
sh(%ebx) := split4B(B0(sh(%eax)), B1(sh(%ebx)),
                    B2(sh(%ebx)), B3(sh(%ebx)))
sh(%eax) := split4B(B0(sh(%eax)), B1(sh(%eax)),
                  B2(sh(%eax)), B3(sh(%eax)))
```

where a `Bn(X)` node represents the extraction of the `n`th byte of `X`.

Note that we split `sh(%eax)` in-place, overwriting the previous unsplit shadow. This is not essential, but if part of `%eax` is used in a subsequent 1-byte operation, having done this we avoid having to split it again. We found that this reduces the number of nodes built by around 15%–25%. Unfortunately, `%eax` may then be used in a 4-byte operation, the node for which will now have an unnecessarily split operand, which makes the DDFG larger and uglier. But we can remove these unnecessary splits in the rewrite stage (see Section 3.8).

Split nodes are fiddly and complicate the implementation significantly. The other possibility would be to shadow every byte of memory separately, and use "merge" nodes for word-sized operations. This would bloat memory use even more, and most operations are done at the word level, so the current approach seems the best.

3.6 Allocating Nodes

Redux manages its own memory pool for allocating nodes. Nodes are allocated sequentially from 1MB superblocks, which is very simple and fast. No garbage collection is done; there has been no need yet, since the programs we have looked at have been small and required only a few megabytes of nodes (Section 6.4 discusses scaling Redux up to bigger programs).

If garbage collection were implemented, reference counting would probably be most suitable, rather than mark-sweep or copy collection. This is because the *root set* during execution is huge—every shadow register, and every shadow memory word—so tracing would be prohibitively slow. Also, there are no cycles to cause problems. We could allocate perhaps three bits in each node to track how many references it has, and the count could saturate at eight; if a node was referenced more than eight times it would never be deallocated. In practice, only a tiny fraction of nodes have this many references.

3.7 Lazy Node Building

One important optimisation reduces the number of nodes built. For each word of static memory initialised at startup, we do not build a node, but instead tag the lowest bit of its shadow. This distinguishes it from ordinary pointers because Redux's allocator ensures nodes are word-aligned, so the bottom two bits of real node pointers are always zero. When reading a node from a shadow word, Redux first checks if the bottom bit is marked, and if so, unmarks the bit and builds a constant node for that word. Thus, nodes are built for static words only when they are used. For Hello World, this avoids building almost 700,000 unnecessary nodes, because a lot of static memory—particularly code—is never used as data.

3.8 Rewriting and Printing

Once the program terminates, Redux performs three passes over the parts of the DDFG reachable from the *root set* of system call nodes. The first pass counts how many times each node is used as an input to another node; those used only once can be rewritten more aggressively. The second pass performs rewriting, mostly making peephole simplifications of the graph that make it more compact and prettier. The third pass prints the graphs.

For the factorial examples in Figure 1, 80 nodes were built for the operations within `main()`, but only nine are shown (the "_exit" node is built outside of `main()`). Outside `main()`, 21,391 nodes were built; Valgrind traces pretty much *everything*, including the dynamic linker (which links functions from shared objects on demand), which account for most of these. By comparison, for the empty C program that just returns zero, Redux builds eight nodes within `main()`, one for the constant zero, and seven book-keeping nodes for building up and tearing down the stack frame, which do not appear in the drawn DDFG.

Redux supports two graph formats. The first is for *dot*, a directed graph drawer that produces PostScript graphs, which is part of AT&T's Graphviz package [5]. The second is for *aiSee*, an interactive graph viewer [1]. We have found that dot's graphs are prettier, but aiSee's interactivity gives more flexibility; for example, the node layout algorithm can be changed. The two programs have very similar input languages, so supporting both is not difficult. All the examples in this paper were drawn by dot.

4 Essences

This section considers possible ways of comparing program equivalence, and shows how programs that perform the same computation in very different ways can have similar or identical DDFGs.

4.1 Program Equivalence

There are various ways to think about whether two programs are "equivalent". At one extreme, if we consider only their visible behaviour, the only important thing is which system calls are made, their order, and what their inputs were. For a given input I, if two programs executed the same system calls in the same order with the same inputs, they they are equivalent with respect to that input I (we will ignore timing issues here).

The other extreme is to consider two programs equivalent with respect to input I if they execute the same instruction sequence when given that input I. This idea of equivalence is so rigid it is almost useless; two programs would probably have to be identical to fit this definition.

Using DDFGs to compare programs gives us a definition of equivalence, based on dataflow, that is somewhere in between these two extremes. We call this level of detail shown in DDFGs the *essence* of a program. Our reason for using this word should hopefully be clear after the following three examples.

4.2 Factorial in C

Figure 1 showed the DDFGs for computing the factorial of five in C, using iteration and naïve recursion. The graphs have the same nodes, because they perform the same operations, but different shapes, because their dataflow is different. However, one can define `fac()` recursively in a way that is equivalent—it has the same dataflow—to the iterative version, by using an accumulator to make it tail-recursive:

```
int faca(int n, int acc) {
   if (n <= 1)
      return acc;
   else
      return faca(n-1, acc*n);
}
```

The graph for this version is identical to that of the iterative version on the left-hand side of Figure 1.

4.3 Factorial on a Stack Machine

Encouraged by this result, we wrote the iterative factorial program in a small stack machine language, for which we had an interpreter. The program, and the resulting DDFG, are shown in Figure 4. We folded the function `read_file()`, which reads the stack machine program from file, and produces the outputs seen in the "f-out" nodes—the integers 5, 1 and 1 which were converted from the ASCII characters '5', '1' and '1' read from the program file (underlined in Figure 4).

It is immediately obvious that the part of the graph computing the factorial is almost identical to those seen previously. The only difference is that instead of using a `decl` x86 instruction to decrement the loop counter, the stack machine program uses a `subl` x86 instruction with 1 as the argument.

Redux sees only pure dataflow; the workings of the interpreter such as the pushes and pops, and loads and stores to memory of intermediate results, are completely transparent. Redux has extracted the essence of the computation which was (almost) identical to that of the C version.

4.4 Factorial in Haskell

Finally, we tried the same experiment in the lazy, purely functional language Haskell [14], using the Glasgow Haskell Compiler [10]. The program and graph are shown in Figure 5. Unlike the previous programs, this one does not compute the factorial of five and return the value. Instead it computes the factorial of five using naïve recursion and accumulator recursion, adds the two results,

```
push 5      dref 0
asg 2       mul
push 1      asg 1
asg 1       dref 0
dref 2      push 1
asg 0       sub
labl 100    asg 0
dref 0      goto 100
push 1      labl 101
psub        dref 1
zero 101    halt
dref 1
```

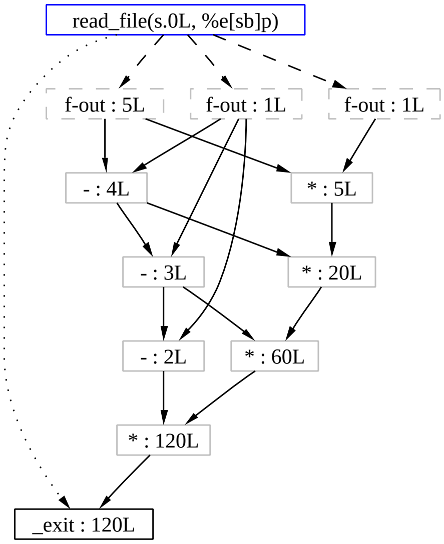


Fig. 4. Iterative `fac(5)` on an interpreted stack machine

and prints the sum. We did this because Haskell programs always return zero to the operating system unless they halt with an exception. We also chose to fold the functions `startupHaskell()` and `shutdownHaskellAndExit()` that initialise and terminate the Haskell runtime system, which consist of calls to the system calls `sigaction()`, `times()`, etc., and are not of interest to us.

The two factorial computations are in the graph's top right-hand corner. They are clearly identical to those in Figure 1. Most of the rest of the graph shows the conversion of the answer 240 into the characters '2', '4' and '0' that are written with `write()`. Once again, despite the program using a very different execution method, the essence of the computation is the same.

5 Possible Uses

DDFGs are very pretty, but it is not immediately obvious how they should be used—they are a solution looking for a problem. This section describes two specific uses that we have implemented, and several other possible uses.

5.1 Debugging

Standard debuggers do not allow backwards execution. This is a shame, because the usual debugging approach is to find a breakpoint after a bug has manifested (e.g. when a variable has an incorrect value), and then repeatedly sets breakpoints earlier in the execution, restarting the program each time, until the erroneous line is identified.

Redux cannot be used for backwards execution, but it does provide a history of all previous computations, and can present that information in a highly readable way. Instead of a invoking a "print" command on an incorrect variable (in memory or a register) at a breakpoint, a user could invoke a "why"

```
main = putStrLn (show (facr 5 + faca 5 1))

facr 0 = 1
facr n = n * facr (n-1)

faca 0 acc = acc
faca n acc = faca (n-1) (acc*n)
```

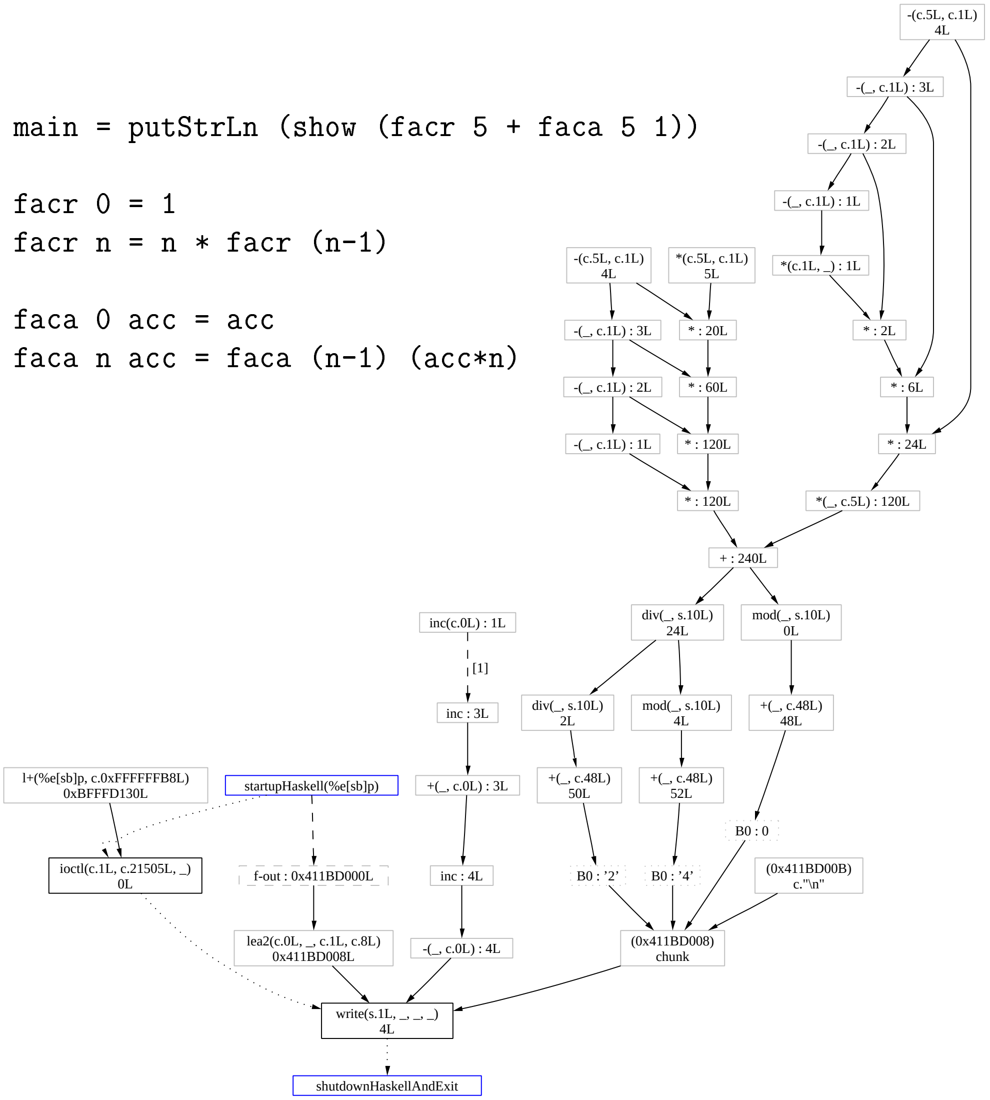


Fig. 5. Iterative and recursive `fac(5)` in Haskell

command that prints out the sub-graph reachable from the specified variable. This graph would show the entire computational history of the variable, which would hopefully make it simple to identify the bug.

As an example of this, when we first implemented the Haskell factorial program in Figure 5, `faca` was defined wrongly. We generated the graph without running the program normally first, and the result for the final "*" node for `faca` was zero instead of 120. Our immediate reaction was to look back at the code to see what the problem was, but we instead looked more carefully at the graph. Tracing back from the erroneous "*" node with value zero, we could see the shape of the computation was identical to that in Figure 1, but that the result of every node was zero. It was instantly obvious that the problem was that we had used zero as the initial accumulator value instead of one. It was also clear that the problem was not, for example, caused by an incorrect number of recursive calls to `faca5`.[7]

Implementing simple support for sub-graph inspection within Redux was quite straightforward—printing a sub-graph for a specific variable part-way through execution is barely different to printing the graph at the program's end. All it required was a way to specify breakpoints. They were added using *client requests*, a Valgrind mechanism that allows the program being supervised to pass a message to a skin. In this case, the client request indicated the address of the variable of interest. This technique requires recompiling the client program to specify a new variable to inspect, which is unfortunate. A better way would be to specify breakpoints via command line arguments, or interactively. This is quite possible within Valgrind by better utilising the debug information, but we have not implemented it yet.

5.2 Dynamic Program Slicing

From debugging via sub-graph inspection, it is a short step to dynamic program slicing [4]. The sub-graph reachable from a variable represents all the operations that contributed to its current value. If we annotate every node with the address of its originating instruction, the addresses of the nodes in that sub-graph form a (non-executable) dynamic data slice [3] for that variable, with respect to the program input. The slice is the "projection" of the sub-graph onto the code.

We did this. Adding instruction addresses to nodes was simple. We printed the slice information in a format readable by `cg_annotate`, a source annotation script that comes with Cachegrind, one of the skins distributed with Valgrind. The following example shows the program slice for the exit status of the iterative factorial C program from Figure 1.

```
 .  .  int fac(int n) {
 1  .     int i, ans = 1;
 .  3     for (i = n; i > 1; i--)
 .  4        ans = ans * i;
 .  .     return ans;
 .  .  }
 .  .
 .  .  int main(int argc, char* argv[]) {
 1  .     return fac(5);
 .  .  }
```

The two numbers on each line indicate how many constant and non-constant nodes originated from it. A '`.`' represents zero. Compare this with the following slice for the stack machine implementation of `fac(5)` from Figure 4.

```
-- line 171 ----------------------------------------
 .    .  if (neg) ++s;
13   18  for(n = 0; isdigit(*s); n = n*10 + *s++ - '0')
```

[7] This example is also notable because Haskell programs are notoriously difficult to debug.

```
  .   .    ;
-- line 242 ---------------------------------------
  .   .   case add:  stk[sp+1] = stk[sp] + stk[sp+1]; ++sp; break;
  .   3   case sub:  stk[sp+1] = stk[sp+1] - stk[sp]; ++sp; break;
  .   .   case psub: r = stk[sp+1] - stk[sp];
-- line 247 ---------------------------------------
  .   .     ++sp;break;
  .   4   case mul: stk[sp+1] = stk[sp+1] * stk[sp]; ++sp; break;
  .   .   case div: r = (int)stk[sp+1] / stk[sp]; stk[sp+1] = r;
```

Most of the nodes arose from the conversion of digit characters to integers, which were hidden in Figure 4 by the folding of `read_file()`. (Note that nodes are built for literal constants each time their containing instruction is executed.) The rest of the nodes come from the actions dealing with the stack machine's `sub` and `mul` instructions. This comparison emphasises just how well Redux can see through a program's book-keeping.

One disadvantage of this approach compared to the sub-graph printing is that not all relevant source code may be available. In our second example, some nodes were not represented because they came from `glibc` code that we did not have the source for.

The most obvious use of this is again for debugging. Sometimes the sub-graph alone might be difficult to interpret, and having a direct connection to the program source might make a variable's computational history much easier to understand. It could also be used in any other way program slicing can be used [15], such as program differencing, software maintenance, or regression testing, but other slicing techniques that have lower overheads might be more appropriate.

5.3 Other Uses

These examples are just a starting point. The following list has several suggestions. Some could not be achieved with Redux in its current form, or might not be feasible in practice, but they give an idea of the range of uses DDFGs might have.

- Program Comprehension: As a generalisation of debugging, DDFGs could be used not to find bugs, but to generally improve understanding of exactly what a program is doing, and where its data flows.
- De-obfuscation: Redux can see through some simple kinds of obfuscation. For example, running a program through some kind of interpreter to hide what it is doing will not be of any use against Redux, as the stack machine interpreter example in Section 4.3 showed. For more highly obfuscated programs, the DDFG could provide a good starting point for understanding what the program is doing. A DDFG might even be useful for analysing cryptographic code in some way.
- Value Seepage: Debugging with sub-graphs considers the past history of a

value. The dual to this is to consider a value's future: how is it used in the rest of the program? Where does it reach? Such information could be useful for understanding programs better, especially security aspects. This is the same idea used in forward slicing [11]. It also has a similar feel to Perl's *taintedness* tracking [17], whereby values from untrusted sources (e.g. user input) cannot be used as is, but must be "laundered" in some way, otherwise a runtime error occurs. Redux would have to be changed to support this, because of the way the graphs are built—each node points back to nodes built in the past, and does not record how its value is used in the future. This change would not be too difficult.

- Program Comparison: Since Redux sees the essence of a program's computation, it could be used to provide a semi-rigorous comparison of programs. This might be useful for determining whether two programs share part of their code, or use the same algorithm to compute something. It could even have implications for software patent issues.[8] The comparisons described in Section 4 are preliminary, but quite promising; this looks like an area worthy of more study.
- Decompilation: The standard approaches to decompilation (e.g. [8,12]) consider only the static program. When a decompiler cannot decompile part of a program in this way, dynamic information such as that in a DDFG might be helpful.
- Limits of Parallelism: Since the DDFG represents the bare bones of a computation, it gives some idea of the level of parallelism in a program's computation, independent of exactly how that computation is programmed. This parallelism is at a somewhat higher level than instruction-level parallelism [16]. Speaking very generally, a wider DDFG implies more inherent parallelism in a program.
- Test Suite Generation: If Redux tracked conditional branches, for each branch that is always or never taken, it might be possible to work back through the DDFG from the conditional test's inputs, and determine how the program's input should be changed so that the branch goes the other way. This could be useful for automatically extending test suites to increase their coverage. While this is a nice idea, in practice it would be very difficult, not least because of the problems Redux has with tracking conditional branches, described in Section 6.3.

6 Difficulties

What Redux does—tracking the entire computational history of a program—is quite ambitious. We have encountered multiple practical difficulties. This section describes some of them, and how we have approached them, with

[8] But we hesitate to enter such a legal swamp.

varying levels of success.

6.1 Normalisation

Since we claim that the DDFG represents the essence of a program's computation, some kind of normalisation should take place, so that small unimportant differences can be ignored. One example we saw in Section 4 was the difference between a `+(X,1)` node and an `inc(X)` node. An x86-specific example is the `lea` instruction, which is often used not for calculating addresses, but as a quasi-three-address alternative to the two-address `add` instruction. Also, the instruction `xorl %reg,%reg` (or `subl %reg,%reg`) is often used to zero a register `%reg`, because it is a shorter instruction than `movl 0x0,%reg`.

Currently, a few transformations are hard-wired into the graph rewriting and printing passes. For example, an `xorl %reg,%reg` node is transformed into a constant node "z.0L" (the 'z' indicates the value comes from a special instruction that always produces zero). A more general approach would be to have some kind of mini-language for specifying transformations of sub-graphs, although we do not yet have a clear idea how to do this.

6.2 Loop Rolling

Redux already does some very basic loop rolling for one case—long chains of repeated "inc" or "dec" nodes. It is important to avoid bloating the drawn graphs with boring chains. This loop rolling is hard-wired into the graph printing phase; no nodes are actually removed from the graph. A small step further is to do the same for chains of nodes that add or subtract the same constant; we have seen this case in graphs for some programs.

A bigger challenge is to find a more general method for rolling up loops. It is not clear to us how to do this; loops with simple dataflow dependencies and few operations are not hard, but the difficulty jumps greatly as the loops grow and/or their dataflow becomes more tangled. Representation of rolled loops is another issue; a good representation is not obvious even for the simple factorial loops in Figure 1.

6.3 Conditional Branches

So far, we have not considered conditional branches at all. Often this is the best thing to do, but sometimes the conditions are critical. Consider this C program:

```
int main(int argc, char* argv[]) {
   if (<complex condition using argc, argv[]>)
      return 1;                 // DDFG: _exit(c.1L)
   else
      return 0;                 // DDFG: _exit(c.0L)
}
```

We ignore the condition, the most interesting part of the program, and end up with a useless graph "_exit(c.0L)" or "_exit(c.1L)".

Unfortunately, only a tiny fraction of conditionals are interesting, and choosing which ones to show is difficult. One way to show them would be to annotate edges with a node representing the branch (or branches) that had to be taken for that edge to be created. Each branch node would have as inputs the values used in the conditional test. In the small example above, the edge between the "c.0L" or "c.1L" node and the "_exit" node would be annotated by a branch node that represents the complex condition. Thus the interesting part of the computation would be included.

To do this properly we must know the "scope" of a conditional, i.e. know which conditional branches each instruction is dominated by. Unfortunately, we do not know how to determine this from the dynamic instruction stream that Redux sees. One possibility would be to modify a C compiler to insert client requests that tell Redux the start and end of conditional scopes. But even if we did this, we fear that the graphs would be bloated terribly by many uninteresting conditions.

6.4 Scaling

The most obvious and pressing problem with Redux is that of scaling. It works well for very small programs, but it is unclear if it will be useful on larger programs. There are two main aspects to this problem.

Firstly, the bigger difficulty is drawing and presenting the graphs. Figure 6 shows two DDFGs for the compression program bzip2 (a 26KB executable when stripped of all symbol information); the left graph is for compressing a two-byte file, the right graph is for compressing a ten-byte file. On a 1400MHz Athlon, running bzip2 under Redux took about 0.8 seconds for both cases, but the graph drawer dot took 8 seconds to draw the first graph, and over two minutes to draw the second graph. The interactive graph viewer aiSee has similar difficulties. The problem is that the graphs are highly connected; long edges between distant nodes slow things down particularly. Presenting the graphs in an intelligible way is also important. The graphs in Figure 6 are already unwieldy.

Much effort has already been put into making the graphs more compact. The `--fold` option, used for Figures 2, 4 and 5, is a good start for mitigating these problems—Figure 5 is four times larger without folding.

Secondly, recording all this information is costly. This is unavoidable to some extent, but there is a lot of room in the current implementation to reduce overhead, by being cleverer with instrumentation to build fewer nodes, adding garbage collection, and so on. The memory space required for nodes, which can be proportional to the running time of the program, could be removed by streaming them to disk. We have not concentrated on this issue yet because the scaling problems of drawing the graphs are more limiting at the moment.

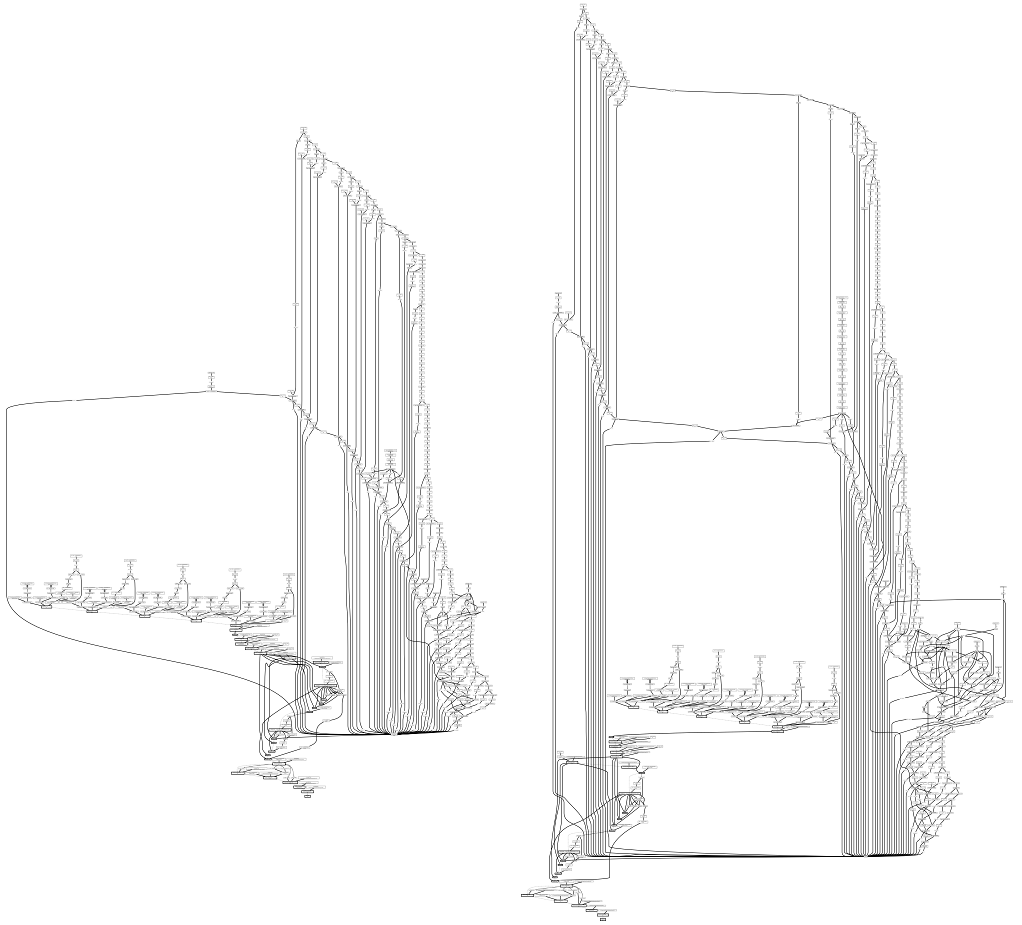


Fig. 6. Compressing a two-byte file and a ten-byte file with bzip2

One likely way to deal with both problems is to be more selective. Currently, all reachable nodes are shown by default, and functions can be folded if the user specifies. Perhaps this should be inverted so that the default is to show a small amount of information, and the user can then specify the interesting parts of the program. Being able to interactively zoom in on parts of the graph would be very useful. We added support for outputting graphs in aiSee format for this reason, because it allows groups of nodes to be folded up. However we have not yet had much chance to experiment with this facility. Alternatively, automatic factoring or compacting of similar sub-graphs would be very useful, if done well; it is hard to say yet how well this could be done.

6.5 Limitations of the Implementation

Because Redux is a prototype, it has not been tried on a great range of programs. It cannot yet handle programs that use floating point arithmetic, nor threaded programs. There are no fundamental reasons for this, it is mostly a matter of time. It also has not been tried on many programs larger than those mentioned, such as bzip2.

7 Related Work

Static dataflow graphs (SDFGs) are commonly used in static analysis of programs. A DDFG is a partial unfolding of a SDFG, but with non-value producing operations (such as assignments) removed, and with control flow "instantiated" in a way that omits all control flow decisions. These differences are crucial; a DDFG omits a lot of information and thus is less precise than an SDFG, which means that its potential uses are very different. For example, the SDFGs for the factorial programs examined in Section 4 would be much more varied than their DDFGs, making them less useful for finding the essence of a program. If we manage to perform loop rolling as described in Section 6.2, the rolled DDFGs will be more similar to SDFGs, but still fundamentally different.

The dynamic dependence graph [4,2] is quite similar to the DDFG, but with additional control nodes and edges. The additional control representation makes the graph a much less abstract representation of a program. Agrawal *et al* used dynamic slicing (but not using the dynamic dependence graph, as far as we can tell) in their Spyder debugger, which annotated code in a way similar to that described in Section 5.1; unlike Redux, Spyder's slices included control statements. Choi *et al* used a similar dynamic graph in their Parallel Program Debugger (PPD) [7].

The most similar tracing tools to Redux we could find are for tracing, visualising and debugging execution of Haskell programs. Hat [6] transforms Haskell programs in order to trace their execution. Three text-based tools can be used to view the traces; one of them, Hat-Trail, allows backwards exploration of a trace. The Haskell Object Observation Debugger (HOOD) [9] is a library containing a combinator `observe` that can be inserted into programs to trace intermediate values, particularly data structures. It also presents the trace information in a text-based way. Both these tools have been inspired by the fact that more conventional debugging techniques are more or less impossible in Haskell because it is lazy.

8 Conclusion

We have seen that Redux is a tool for drawing dynamic dataflow graphs of programs. These graphs show the computational history of the program, and reduce programs to a minimal *essence*; programs that compute the same thing in multiple ways have identical or very similar graphs. The uses of Redux are not completely clear, although we discussed how it could be used for debugging and program slicing, as well as more speculative uses. We also discussed challenges and difficulties of the implementation.

We feel that DDFGs have great potential, because they get to the heart of what programs are computing. We hope that they can be used in a range of ways for understanding, debugging, and improving programs.

Acknowledgement

Many thanks to: Julian Seward, for creating Valgrind; and Harald Søndergaard and Peter Stuckey, for the use of their stack machine interpreter. The first author gratefully acknowledges the financial support of Trinity College, Cambridge.

References

[1] AbsInt Angewandte Informatik GmbH. aiSee – graph visualisation. `http://http://www.absint.com/aisee/`.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in presence of unconstrained pointers. In *Proceedings of TAV4*, pages 60–73, Victoria, British Columbia, Canada, Oct. 1991.

[3] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience*, 23(6):589–616, June 1993.

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of PLDI'90*, pages 246–256, White Plains, New York, USA, June 1990.

[5] AT&T Labs-Research. Graphviz. `http://www.research.att.com/sw/tools/graphviz/`.

[6] O. Chitil, C. Runciman, and M. Wallace. Transforming haskell for tracing. In *Proceedings of IFL 2002*, Madrid, Spain, Sept. 2002. To appear.

[7] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, Oct. 1991.

[8] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Faculty of Information Technology, Queensland University of Technology, Australia, July 1994.

[9] A. Gill. Debugging haskell by observing intermediate data structures. In *Proceedings of the 2000 Haskell Workshop*, Montreal, Canada, Sept. 2000.

[10] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[12] A. Mycroft. Type-based decompilation. In *Proceedings of ESOP'99*, volume 1576 of *LNCS*, pages 208–223, Amsterdam, The Netherlands, Mar. 1999.

[13] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of RV'03*, Boulder, Colorado, USA, July 2003. To appear.

[14] S. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[15] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[16] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, Digital Western Research Laboratory, Palo Alto, California, USA, Nov. 1993.

[17] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reilly, 3rd edition, 2000.