Faster production of redex trails: The Hat G-Machine

Tom Shackell and Colin Runciman
Tom.Shackell@cs.york.ac.uk, Colin.Runciman@cs.york.ac.uk

University of York, UK

Abstract

The Hat system provides a method for tracing a lazy functional program. However Hat works by transforming the source program into a much larger self-tracing variant that runs between 15 and 100 times slower and uses several times as much memory. We show how equivalent traces can be generated much more efficiently by modifying the underlying abstract machine. Our approach shows that it is only necessary to add a few extra machine instructions and change the interpretation of a few others in order to generate Hat traces efficiently.
Keywords: Tracing, Haskell, lazy functional programming, abstract machines.

1 INTRODUCTION

It is often said that an advantage of functional languages is that they make it more likely that programmers will write correct code. However, even functional programs sometimes have bugs. In order to correct them it is often necessary to know why the program gave a particular answer. It can also be useful to understand why a program gave a particular answer for reasons other than debugging, for example, in order to gain a better understanding of the program. This can be particularly important for people modifying code that they did not write.

One method to debug and understand a program is to produce a trace of everything that the program did. Such an approach is used in systems such as Hat [10] as well as in algorithmic debugging systems such as Freja [4] and Buddha [6]. This paper will look principally at the Hat system.

Hat produces traces in way that is portable across different compilers, but the traced computation uses much more memory and typically runs between 15 and 100 times slower than the untraced case. This paper describes how the same traces can be produced much more efficiently by modifying the underlying abstract machine.

1.1 The difficulties of tracing lazy functional languages

The method traditionally used for tracing and debugging in strict functional and imperative languages is to step through the execution sequentially and examine the runtime state of the system, such as the contents of the heap and values of variables on the stack [9]. Such an approach is less effective for lazy functional languages,

however, as the order of evaluation is demand driven and so is often un-intuitive to the programmer. In a lazy language the execution of a piece of work is suspended until its evaluation is necessary in forming the output of the program.

Producing a complete trace of the execution of the program means that debugging becomes a post-mortem activity. The advantage of this approach is that it allows the execution to be explored independently of evaluation order. It also means that the runtime state of the program cannot be altered during the tracing process, which is in keeping with the declarative nature of functional programs.

Hat works by using source level transformation: the original Haskell program is translated into a new Haskell program. When this new program is evaluated it does the same computation as the original but as a side effect also produces a trace of the evaluation. Such an approach has the advantage of being portable across different compilers and platforms. However, it also makes the source code several times larger, increases compile times, can greatly increase the memory usage and can greatly reduce the runtime performance of the program. For some large applications such a performance loss makes tracing impractical.

1.2 Tracing using a modified abstract machine

An alternative approach to transforming the program source is to use a modified compiler and abstract machine. The program is compiled with additional debugging information and then as execution proceeds the abstract machine generates a trace of the computation. Such an approach is not portable across different compilers but it has the advantage that it has much less interpretive overhead when compared with tracing by program transformation.

2 THE G-MACHINE

The G-Machine is an abstract machine for implementing lazy functional languages by graph rewriting and is described by Augustsson and Johnsson in [3]. Suspended computations are represented in the heap as graphs. When the value of the suspended computation becomes necessary the function described by the graph is executed and the graph is rewritten with its result. This process continues until the final result of the computation has been obtained.

This paper describes modifications to the nhc compiler [8], which is implemented based on the Spineless G-Machine [1].

The definition for a function is compiled into a series of SG-Machine instructions. When a closure is evaluated the SG-code builds a graph representing the body of the function in the heap and then returns and updates the closure.

For example, consider the following definition of a function:

```
unary :: Int -> [Int]
unary 0 = []
unary n = 1 : unary (n-1)
```

```
FUNCTION unary(n):
  PUSH_ARG n              [ n ]
  EVAL                    [ n ]
  CASE                    [ ]
     0  -> PUSH_GLOBAL []  [ [] ]
           RETURN
     _  -> PUSH_INT 1      [ 1 ]
           PUSH_ARG n      [ n, 1 ]
           MKAP (-) 2      [ n-1 ]
           MKAP unary 1    [ unary (n-1) ]
           PUSH_INT 1      [ 1, unary (n-1) ]
           MKCON (:)       [ 1 : unary (n-1) ]
           RETURN
```

FIGURE 1. The SG-Machine instructions for the `unary` function.

Figure 1 shows the SG-Machine instructions for the `unary` function with the values on the stack during evaluation are shown on the right.

The `RETURN` instruction updates the original `unary 2` closure with the result `1 : unary (2-1)`. If the same `unary 2` closure is evaluated again then instead of recomputing the graph, the result is returned immediately.

Further computation might cause the `unary (2-1)` closure to be evaluated leading to more computation and graph generation. A chain of such applications and updates is shown in Figure 2.

3 THE HAT TRACE STRUCTURE

The Hat trace is a graph that provides a detailed record of the graph reduction process when a program is executed. A trace can be viewed by the Hat tools to examine what the computation did [2]. The trace principally records applications of functions to particular arguments. The trace also records the parent application, in whose body this one was created, and what the result of this application is.

Figure 3 shows the complete trace of the `unary 2` example. The dotted arrows show the parent of a node and the dot-dashed arrows show the updating of results.

The Hat trace is a graph structure stored as a series of nodes in a `file`. A reference to a node is given by the `file` position of that node.Nodes store references to other nodes by storing their `file` positions. Figure 4 shows the basic structure of an Hat trace, formalised as a set of concrete Haskell types:

When a redex is created its result is *Unevaluated*. If that redex is evaluated then the redex is updated to *Entered*, and when its result is available it is updated with a *Ref* to the result.

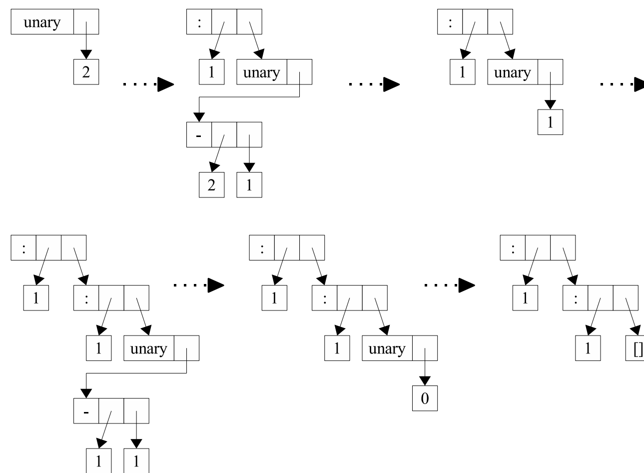

FIGURE 2. A sequence of graph reductions for `unary 2`.

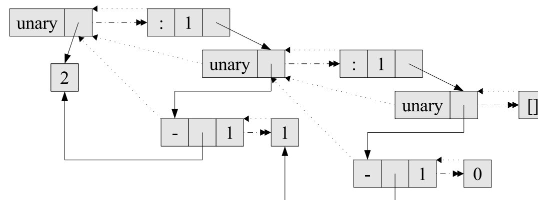

FIGURE 3. The Hat trace for the `unary 2` example.

Ref	a direct reference to a file position.
NoRef	signifies that there is no useful reference, for example the parent of a top-level function.
Entered	is used as the result reference for nodes that have been evaluated but not updated with a result yet.
Interrupted	shows computations that were stopped by the user.
Unevaluated	shows nodes that have not yet been evaluated.

4 MODIFYING THE SG-MACHINE FOR TRACING

The correspondence between the Hat traces and the graph rewriting semantics strongly suggests that a Hat trace could be generated by a modified SG-Machine. The basic idea is that when new nodes are built in the heap a corresponding node is built in the trace. Similarly whenever a heap nodes is updated with a result, the corresponding trace node is updated with the corresponding result.

```
data Ref a  = NoRef              | Ref FilePos
            | Entered            | Interrupted
            | Unevaluated
data Expr   = Ap       { fun    :: Ref Expr
                         args   :: [Ref Expr],
                         parent :: Ref Expr,
                         src    :: Ref SrcPos,
                         result :: Ref Expr }
            | Atom     { atom   :: Ref Atom,
                         parent :: Ref Expr,
                         src    :: Ref SrcPos,
                         result :: Ref Expr }
            | Case     { cond   :: Ref Expr,
                         parent :: Ref Expr,
                         result :: Ref Expr }
            | If       { cond   :: Ref Expr,
                         parent :: Ref Expr,
                         result :: Ref Expr }
            | Forward { result :: Ref Expr }
            | Hidden  { parent :: Ref Expr,
                         result :: Ref Expr }
data Atom   = Id String SrcPos | IntVal Int
            | CharVal Char     | ...
```

FIGURE 4. The basic Augmented Redex Trail structure.

4.1 The trace stack

Although the trace of a function closely matches its representation in the heap there are things recorded in the trace which have no representation in the heap, for example case expressions. In order to accommodate these features an extra stack is added to the abstract machine, the trace stack.

Consider the `max` function, which uses guards.

```
max :: Int -> Int -> Int
max x y
  | x > y     = x
  | otherwise = y
```

The code for tracing the `max` function is shown in Figure 6 — the untraced code is the same without the 'T's in front of the instructions.

Figure 5 compares the heap graph for `max 4 3` with the desired trace graph. The SG-Machine builds a graph for `4 > 3` and evaluates it to `True`. It then evaluates the guard, pushes `4` on the stack and updates the `max` closure with the result on the top of the stack, which is `4`. In the trace, however, the `max` trace node is updated with a guard and the guard is updated with the trace for `4`. When `if`, `guard` and `case` are introduced the trace structure no longer corresponds exactly with the heap structure.

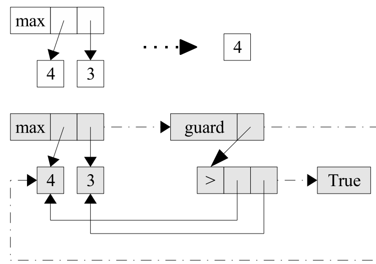

FIGURE 5. The graph reduction and Hat trace for `max 4 3`

```
FUNCTION max(x,y)  [ ]         < max x y >
  TPUSH_ARG y      [ y ]       < y, max x y >
  TPUSH_ARG x      [ x, y ]    < x, y, max x y >
  TMK_AP (>) 2     [ x > y ]   < x > y, max x y >
  TEVAL            [ x > y ]   < x > y, max x y >
  TGUARD           [ ]         < | x > y >
     TPUSH_ARG x   [ x ]       < x, | x > y >
     TEVAL         [ x ]       < x, | x > y >
     TRETURN
  ELSE             [ ]         < | x > y >
     TPUSH_ARG y   [ y ]       < y, | x > y >
     TEVAL         [ y ]       < y, | x > y >
     TRETURN
```

FIGURE 6. The Hat SG-Machine instructions for the `max` **function.**

These issues are solved by use of the trace stack. Instructions in the G-Machine change the graph corresponding to the normal stack and change the trace corresponding to the trace stack. In Figure 6, the trace stack is shown on the far right of the figure.

4.2 Semantics

Adapting the SG-Machine for tracing required some modification to existing instructions as well as the addition of new instructions. The semantics for the new and modified instructions are shown in Figure 7. Only the rules for SG-Machine instructions which have been modified for tracing are shown.

We use `x` **update** `y` to notate that the trace node `x` should have its result field updated with `y`.

4.3 Implementation

The back end of the nhc compiler has to be modified to generate the new instructions and to add extra information such as function names and source code po-

The state is an 8 tuple:

$<$ instrs, node, parent, stack, tstack, heap, dump, trace $>$

intrs	list of instructions left to execute
node	the heap address of the heap node currently being evaluated
parent	the trace reference of the current parent
stack	the program stack, a list of heap addresses
tstack	the trace stack, a list of trace references
heap	a function from heap addresses to pairs of heap nodes and trace references.
trace	a function from trace references to trace nodes
dump	a tuple of (code, node, parent, stack, tstack) for recursive calls

$<$ TPUSH_ARG n : cs, v, p, as, ts,
h[v $\mapsto$ (AP(f, x_0 .. x_m),_), $x_n \mapsto$ (_, t)],
tr, ds $> \Rightarrow$
$<$ cs, v, p, x_n : as, t : ts, h, tr, ds $>$

$<$ TPUSH n : cs, v, p, a_1 .. a_n : as, ts,
h[$a_n \mapsto$ (_,t)], tr, ds $> \Rightarrow$
$<$ cs, v, p, a_n : a_1 .. a_n : as, t : ts, h, tr, ds $>$

$<$ TPUSH_INT n : cs, v, p, as, ts, h, tr, ds $> \Rightarrow$
$<$ cs, v, p, a : as, t : ts,
h $\oplus$ { a $\mapsto$ (CON(n),t) },
tr $\oplus$ { t $\mapsto$ INT(p, Unevaluated, n) }, ds $>$

$<$ TIF xs ys : cs, v, p, a : as, t_0 : t_1 : ts,
h[a $\mapsto$ (CON(True), _)], tr, ds $> \Rightarrow$
$<$ xs $+\!\!+$ cs, v, p, as, u : ts, h,
tr $\oplus$ { u $\mapsto$ IF(p, Entered, t_0),
t_1 **update** Result u }, ds $>$

$<$ TIF xs ys : cs, v, p, a : as, t_0 : t_1 : ts,
h[a $\mapsto$ (CON(False), _)], tr, ds $> \Rightarrow$
$<$ ys $+\!\!+$ cs, v, p, as, u : ts, h,
tr $\oplus$ { u $\mapsto$ IF(p, Entered, t_0),
t_1 **update** Result u }, ds $>$

$<$ TMK_AP f n : cs, v, p, a_1 .. a_n : as,
t_1 .. t_n : ts, h, tr, ds $> \Rightarrow$
$<$ cs, v, p, a : as, t : ts,
h $\oplus$ { a $\mapsto$ (AP(f, a_1 .. a_n),t) },
tr $\oplus$ { t $\mapsto$ (AP(p, Unevaluated, f, t_1 .. t_n) },
ds, $>$

$<$ TMK_CON c n : cs, v, p, a_1 .. a_n : as,
t_1 .. t_n : ts, h, tr, ds $> \Rightarrow$
$<$ cs, v, p, a : as, t : ts,
h $\oplus$ { a $\mapsto$ (CON(c, a_1 .. a_n),t) },
tr $\oplus$ { t $\mapsto$ (CON(p, Unevaluated, c, t_1 .. t_n) },
ds, $>$

$<$ TEVAL : cs, v, p, a : as, t : ts,
h[a $\mapsto$ (AP(f, a_0 .. a_n), u)], tr, ds $> \Rightarrow$
$<$ code f, a, t, [], [t],
h $\oplus$ { a $\mapsto$ (HOLE(f, a_0 .. a_n), u) }
tr $\oplus$ { t **update** Entered }, ds, $>$

$<$ TEVAL : cs, v, p, a : as, ts,
h[a $\mapsto$ (CON(_), _)], tr, ds $> \Rightarrow$
$<$ cs, v, p, a : as, ts, h, tr, ds $>$

$<$ TRETURN : cs, v, p, a : as, t_0 : t : ts, h,
tr, (cs', v', p', as', ts') : ds $> \Rightarrow$
$<$ cs', v', p', a :as', t : ts',
h $\oplus$ { v $\mapsto$ (IND(a),_) },
tr $\oplus$ { t **update** Result t_0 }, ds $>$

$<$ TAPPLY n : cs, v, p, a_0 : a_1 .. a_n : as,
t_0 : t_1 .. t_n : ts, h[$a_0 \mapsto$ (AP(f, x_1 .. x_m), t)],
tr, ds $> \Rightarrow$
$<$ TAPPLY' (z-m) : cs, v, p, a_0 : a_{z-m} .. a_n : as,
u:ts,
h $\oplus$ { $a_0 \mapsto$ (AP(f, x_1 .. x_m $+\!\!+$ a_1 .. a_{z-m}), t) },
tr $\oplus$ { u $\mapsto$ AP(p, t_0, t_1 .. t_n) }, ds $>$
where
z = arity f

$<$ TAPPLY' n : cs, v, p, as, ts, h, tr, ds $> \Rightarrow$
$<$ cs, v, p, as, ts, h, tr, ds $>$
when n $\leq$ 0

$<$ TAPPLY' n : cs, v, p, a : b : as, ts, h, tr, ds $> \Rightarrow$
$<$ TAPPLY' (n-1) : cs, v, p, c : as, ts,
h $\oplus$ { c $\mapsto$ (AP(apply, [a,b]), NoRef) }
tr, ds $>$

FIGURE 7. Semantics for the modified SG-Machine instructions.

sitions. The most extensive modifications are in the nhc interpreter. Necessary modifications are:

- changing the heap node format: every heap node gaines 2 additional fields. The first holds information about each node such as its name and module, as well as flags about whether this node is already traced or not. The second holds a reference to source position information for untraced nodes, or a trace reference to the trace node for traced nodes.
- adding a trace stack: an additional stack is needed to accommodate certain features of the trace, see Section 4.1.
- adding code for the new instructions and modifying existing instructions, see Section 4.2.
- introducing code for writing Hat traces from the hat-trans library.

4.4 Dealing with compiler introduced constructs

One issue that substantially complicates tracing at the abstract machine level is that the program trace should correspond to the program source. This issue is easily solved in the Hat-trans system because it operates at a syntactic level. However the issue is more complicated in the Hat G-Machine case because the translation from Haskell source to SG-Machine instructions introduces constructs which are not in the original source. There are three main instances of this caused by pattern matching, lambda lifting and primitive functions.

Pattern matching might be compiled into one or more case statements in the SG-Machine instructions. These case statements obviously have no corresponding aspect in the source code, and so they should not be traced. The solution is to remember which case statements were in the original source and which were introduced by the compiler, and then compile 'real' case statements to be traced and 'introduced' statements without tracing.

Lambda lifting introduces new top-level definitions, which also should not be traced. The solution is to use a `TFORWARD` instruction, which traces the application on the top of the program stack with a simple forward node. The forward node is later updated with the result of the lifted function.

Some functions to manipulate primitive types, such as addition and subtraction, are implemented using primitive operations. For example the SG-Machine instruction to add two integers is `ADD_I`. These instructions complicate tracing because they need to be recorded as function applications in the trace although they are not implemented as function applications in the SG-Machine. This is resolved by having an additional `TPRIMITIVE` instruction that traces a primitive as though it were function application.

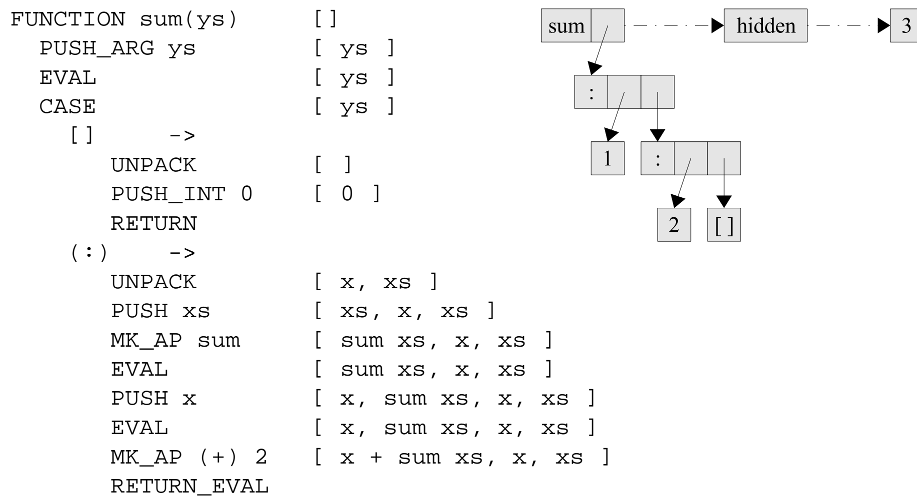

FIGURE 8. The SG-Machine instructions and trace for the trusted `sum` function.

4.5 Trusting

A complete trace of a program can be very large, often running into hundreds of megabytes. In order to reduce this problem Hat has the concept of trusting. Trusted functions are functions whose implementations are assumed to be correct and thus their internal workings do not need to be traced. For example, all the functions in the standard Haskell prelude are considered trusted.

The definition for the standard `sum` function is shown below and the SG-Machine instructions are shown in Figure 8.

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Figure 8 also shows the trusted computation of `sum [1,2]`. The initial application to sum, including its arguments, is traced. The result of this application is 'Hidden' showing that it was computated inside a trusted application. The instructions for `sum [1,2]` calls `sum [1]`, which in turn calls `sum []`, neither of the recursive applications is shown in the trace because they are considered internal details of the `sum` function.

The three applications `sum [1,2]`, `sum [1]` and `sum []` all return results; however, only the result of the root of the trusted computation, `sum [1,2]` is shown in the trace, because `sum [1,2]` has a corresponding node in the trace whereas the others do not.

In general we can say:

- applications to trusted functions from untrusted computations are recorded, and when the computation produces a result that result is recorded.

- applications to trusted functions from trusted computations are not recorded, and the result of that application is not be recorded.

- applications to untrusted functions from trusted computations are recorded, and their evaluation is traced as described in Section 4.1.

The mechanism used for tracing untrusted functions cannot easily be extended to deal with trusted ones so a different mechanism is used. Although the internal details of a trusted function should not be traced, the result of an application to a trusted function might be needed in the trace. The basic principle is to delay tracing until it is deemed absolutely necessary. Implementing this principle involves modification to three SG-Machine instructions: EVAL, RETURN and RETURN_EVAL.

- The EVAL instruction evaluates applications to weak head normal form. It is modified so that when an untrusted function is evaluated from within a trusted one the untrusted application is traced.

- The RETURN instruction is used when computation is returning a WHNF result. It is modified so that when a trusted function that has been traced returns a result that result is partially traced.

- RETURN_EVAL is used to return a non-WHNF result from a function. It is modified so that when a traced and trusted function returns and evaluates another trusted function then the responsibility for updating the trace is passed from the first function to the next.

4.5.1 The EVAL instruction

The EVAL instruction is used when an application should be reduced to a weak head normal form (WHNF). When EVAL is called inside a trusted computation one of three things can happen: if the object being evaluated is an application to an untrusted function, then the application is traced recursively, as in Figure 9; if the object being evaluated is an application to a trusted function in which case no trace is made; if the object being evaluated is already in WHNF then no computation occurs and nothing is traced.

Forward nodes act like indirections in the trace. An untraced trusted application is traced with a *Forward* because the application must be an internal detail of a trusted function. Since trusted applications are an internal detail there is no need to trace their applications; however, their result may be required. So when a trusted applications is evaluated the *Forward* is updated with the trace for its result.

4.5.2 The RETURN instruction

When a result is returned using a RETURN instruction inside a trusted computation there are two possibilities. If the node under evaluation is traced, then the returned heap node is traced recursively and the trace of the node currently under evaluated

```
traceRec :: Node -> TraceRef
traceRec node
   | isTraced node    = traceOf node
   | isUntrusted node = makeApplication node args
   | isConstr node    = makeConstr node args
   | isTrusted node   = makeForward
   where
   args = map traceRec (argsOf node)
```

FIGURE 9. Haskell pseudocode for recursively tracing a heap node

is updated with the traced of the result. If the node under evaluation is not traced then no special action is taken.

4.5.3 The RETURN_EVAL instruction

RETURN_EVAL is used to return a non-WHNF result from a function. In common with the RETURN instruction there are two possibilities.

If the heap node under evaluation is not traced then the heap node is updated with the result heap node, and the result node is evaluated as normal. This evaluation might cause further tracing. For example, if the result node is an application to an untrusted function.

If the heap node under evaluation is traced and the result heap node is not traced then the trace of the result node is set to be the trace of the node under evaluation, and the result node is evaluated as normal. The responsibility of updating the trace node has been passed to the result application.

If the heap node under evaluation is traced and the result heap node is also traced then the trace node of the heap node under evaluation is updated with the trace node of the result heap node.

5 RESULTS

The performances of Untraced nhc, Hat G-Machine and Hat-trans are compared using the following programs from the no-fib benchmark suite [1].

- Queens: The N-Queens program, where n = 8.
- Wheel1: A program to list the first n primes, where n = 1000.
- Primes: A naïve implementation for finding the nth prime using the sieve of Eratosthenes, where n = 500.

5.1 Compile-time results

The compile-time results are shown in the first part of Table 1. Results are shown for two indicators: the size of the final binary in Kb and the time to compile the

[1] http://www.dcs.gla.ac.uk/fp/software/ghc/nofib.html

program in seconds.

Table 1 shows that the object code size for the Hat G-Machine is always within a factor of 3 of the untraced case, which is substantially smaller than the code generated by Hat-trans.

The compiler adds information to each function such as the name of the function and the source code positions for each application, which expands the binary size considerably. However the size is still much better than Hat-trans, which as well as adding the same information also adds space for all the combinators that it uses.

The compile time for the Hat G-Machine is also very much improved compared to Hat-trans. Compile time results for the Hat G-Machine are not substantially different to the untraced case, with the slight extra cost coming from having to write a bigger binary. In comparison Hat-trans has the overheads of a translation phase and must compile a much larger Haskell program.

5.2 Run-time results

Run-time results are shown in the second part of Table 1. The results are shown for a number of key performance indicators.

Mem	peak live heap memory after a garbage collection in Kb
Time	complete program runtime in seconds
GCs	number of garbage collections

The memory usage of Hat G-Machine programs is always within a factor of two of the untraced program; in comparison programs generated by Hat-trans use between 3 and 12 times the memory needed in the untraced case. In terms of runtime performance the Hat G-Machine is between a factor of two and four slower than the untraced program; programs generated by Hat-trans typically being around a factor of 50 slower.

The extra memory used by the Hat G-Machine is two extra words for every heap cell. Hat-trans works by wrapping every constructor and function application in a data structure, it also wraps every function call in various tracing combinators, which adds a lot of extra overhead.

The principal additional overheads in the Hat G-Machine are the overhead of file IO and the additional instructions managing the trace stack. Hat-trans has the overheads of the tracing combinators and the wrapping and unwrapping functions.

5.3 Trace-time results

Are the traces produced by Hat-trans and the Hat G-Machine the same? No, there are differences in the traces of trusted functions, indeed the Hat G-Machine's handling of trusting is more accurate. So we cannot just test traces for equality, but are they in some sense equivalent?

TABLE 1. Performance results

Program	Compile time		Run time		
	Object Code(Kb)	Time(s)	Mem(Kb)	Time(s)	GCs
Primes-NT	282	0.47	3	0.88	121
Primes-HGM	709	0.53	6	1.91	220
Primes-HT	1753	10.43	20	66.92	9065
Queens-NT	207	0.49	8	0.17	27
Queens-HGM	488	0.55	14	0.61	48
Queens-HT	1755	10.80	103	8.86	1147
WheelSieve-NT	283	0.54	7	0.16	22
WheelSieve-HGM	713	0.66	13	0.47	41
WheelSieve-HT	1764	11.20	23	9.30	1234

TABLE 2. Significant variations in hat-observe application counts for N-Queens

Function	Count HGM	Count HT
$+$	279	58933
$-$	8	29335
$/=$	0	87981
$==$	360	369
fromInteger	0	41768
unlines	1	0

Application Counts

The hat-observe tool provides a way to test a looser equivalence about a trace such as the numbers of applications of functions. If we do this for the N-Queens program, the application counts are indeed identical for all functions defined in the Main module. But the counts for some Prelude functions differ as shown in Table 2.

- The slight discrepancy between the numbers of applications of the `==` function is because Hat-trans translates some case expressions over integers into guarded patterns.
- Applications to `fromInteger` don't appear in the source program, but they are introduced by the Hat-trans translation.
- The missing application to `unlines` in the Hat-trans trace comes from taking a different design decision on the handling of constant applicative forms (CAFs).

The `unlines` function is defined as a partial application of `foldr`. So internally `unlines` isn't applied to the argument given, instead it is evaluated, returning a partial application and that is applied to the argument. Hat-trans does not record the reduction of such trusted CAFs as the result is a trusted internal detail.

However this could be confusing to the user, who might expect to see an application of the `unlines` function. So the Hat G-Machine records the application to `unlines`.

- The large differences in the numbers of applications of `+`, `-` and `/=` are due to the fact that the Hat G-Machine currently does not handle dictionaries. There is a question of whether having to generate these applications would undermine the performance figures: this does not appear to be the case as the N-Queens program with the overloading removed, has identical performance figures to the ones shown in Table 1.

6 RELATED WORK

The most closely related work is that of Henrik Nilsson in his Freja system [4, 5]. Like the Hat G-Machine, Freja generates a trace of a lazy functional language by modifying the abstract machine. However there are a number of key differences: Freja generates simpler EDT structures rather than the richer Hat trace structure; Freja generates the traces 'piecemeal' in memory rather than in a file as Hat does; Freja is based on the standard G-Machine rather than the spineless G-Machine of nhc.

Many of the same ideas are present. For example Nilsson also augments the heap nodes with additional information, although the information Freja stores is quite different. Freja also has equivalents to instructions such as `TMK_AP`.

Freja, however, does not generate traces for If, Case or Guard nodes and as such does not have any concept of a trace stack. Freja does have the concept of trusting; however, it is achieved by simply disabling tracing temporarily. It doesn't deal with the issue of wanting to know the result of a trusted computation but not wanting to see the internal details, as in the context of algorithmic debugging the result of a trusted function provides no new information.

The other work most similar is Pope's Buddha system [7] which, like Hat-trans, is based principally on program transformation. Like Freja, Buddha generates EDT structures. Buddha uses a single un-portable function to get the internal representation of a Haskell value, enabling it to look at a value without evaluating it. Buddha is not based on modifying an abstract machine.

7 CONCLUSIONS AND FURTHER WORK

7.1 Conclusions

The work on the Hat G-Machine has shown that by making minor changes to the SG-Machine code generated, and by extending the interpretation of SG-Machine code, it is indeed possible to generate Hat traces. With some additional effort this scheme can be extended to deal with the computations over trusted functions.

The expected efficiency gains from tracing at the abstract machine level are indeed realised. The Hat G-Machine produces traces more than an order of magnitude faster than corresponding programs generated by Hat-trans, with only a 2 to 4 fold increase in execution time compared to untraced code. It also substantially reduces the memory consumption, compile time and object code size when compared to the Hat-trans approach.

The Hat G-Machine and Hat-trans handle trusting differently and in general the traces generated by the Hat G-Machine are more accurate in this respect. The work may also lead to certain design decisions taken in Hat-trans being re-evaluated.

7.2 Further work

The unwanted differences between the traces produced by Hat-trans and those produced by the Hat G-Machine need to be resolved. It will be necessary to characterise more precisely the expected correspondence between the traces generated by the two methods. The traces generated by the Hat G-Machine have not yet been checked extensively against the full range of viewing tools.

In due course it is our aim to merge the Hat G-Machine work into the released nhc98 system.

ACKNOWLEDGEMENT

The first author is supported by a PhD studentship from the Engineering and Physical Sciences Research Council, UK.

REFERENCES

[1] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-Machine. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 244–258. ACM Press, 1988.

[2] K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, number 2638 in LNCS, pages 59–99, Oxford, August 2003.

[3] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[4] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linkoping University, April 1998.

[5] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *J. Funct. Program.*, 11(6):629–671, 2001.

[6] B. Pope and L. Naish. Practical aspects of declarative debugging in Haskell 98. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 230–240, New York, NY, USA, 2003. ACM Press.

[7] B. Pope and L. Naish. A program transformation for debugging Haskell 98. In *CRIPTS '16: Proceedings of the twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, pages 227–236, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[8] N. Röjemo. Highlights from nhc: a space-efficient Haskell compiler. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 282–292. ACM Press, 1995.

[9] A. P. Tolmach. *Debugging standard ML*. PhD thesis, Princeton, NJ, USA, 1992.

[10] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In R. Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).