Supero: Making Haskell Faster

Neil Mitchell and Colin Runciman

University of York, UK, http://www.cs.york.ac.uk/~ndm

Abstract. Haskell is a functional language, with features such as higher order functions and lazy evaluation, which allow succinct programs. These high-level features are difficult for fast execution, but GHC is a mature and widely used optimising compiler. This paper presents a whole-program approach to optimisation, which produces speed improvements of between 10% and 60% when used with GHC, on eight benchmarks.

1 Introduction

Haskell [15] can be used in a highly declarative manner, to express specifications which are themselves executable. Take for example the task of counting the number of words in a file read from the standard input. In Haskell, one could write:

```
main = print ∘ length ∘ words =<< getContents
```

From right to left, the getContents function reads the input as a list of characters, words splits this list into a list of words, length counts the number of words, and finally print writes the value to the screen.

An equivalent C program is given in Figure 1. Compared to the C program, the Haskell version is more concise and more easily seen to be correct. Unfortunately, the Haskell program (compiled with GHC) is also three times slower than the C version (compiled with GCC). This slowdown is caused by several factors:

Intermediate Lists The Haskell program produces and consumes many intermediate lists as it computes the result. The getContents function produces a list of characters, words consumes this list and produces a list of lists of characters, length then consumes the outermost list. The C version uses no intermediate data structures.

Functional Arguments The words function is defined using the dropWhile function, which takes a predicate and discards elements from the input list until the predicate becomes true. The predicate is passed as an invariant function argument in all applications of dropWhile.

Laziness and Thunks The Haskell program proceeds in a lazy manner, first demanding one character from getContents, then processing it with each of the functions in the pipeline. At each stage, a lazy thunk for the remainder of each function is created.

```
int main()
{
        int i = 0;
        int c, last_space = 1, this_space;
        while ((c = getchar()) != EOF) {
                this_space = isspace(c);
                if (last_space && !this_space)
                        i++;
                last_space = this_space;
        }
        printf("%i\n", i);
        return 0;
}
```

Fig. 1. Word counting in C.

Using the optimiser developed in this paper we can eliminate all these overheads. We obtain a program that performs *faster* than the C version. The central idea of the optimiser is to evaluate as much of the program as possible at compile time, leaving a residual program consisting only of actions dependent on the input data.

Our goal is an automatic optimisation that makes high-level Haskell programs run as fast as low-level equivalents, eliminating the current need for hand-tuning and low-level techniques to obtain competitive performance. We require no annotations on any part of the program, including the library functions.

1.1 Roadmap

We first introduce a Core language in §2, on which all transformations are applied. Next we describe our optimisation method in §3. We then give a number of benchmarks, comparing both against C (compiled with GCC) in §4 and Haskell (compiled with GHC) in §5. Finally, we review related work in §6 and conclude in §7.

2 Core Language

All our optimisations operate on a standard Core language, documented in [6]. The expression type is given in Figure 2. A program is a mapping of function names to expressions. Our Core language is higher order and lazy, but lacks much of the syntactic sugar found in Haskell. Pattern matching occurs only in case expressions, and all case expressions are exhaustive. All names are fully qualified. Haskell's type classes have been removed by the dictionary transformation [24].

The Yhc compiler, a fork of nhc [20], can output Core files. Yhc can also link in all definitions from all required libraries, producing a single Core file representing the whole program.

expr $= v$	variable
$\mid c$	constructor
$\mid f$	function
$\mid x\ y$	application
$\mid \lambda v \rightarrow x$	lambda abstraction
$\mid$ **let** $v = x$ **in** y	let binding
$\mid$ **case** x **of** $\{p_1 \rightarrow y_1; ...; p_n \rightarrow y_n\}$	case expression

pat $= c\ \overrightarrow{vs}$

Where v ranges over variables, c ranges over constructors, f ranges over functions, x, y and z range over expressions and p ranges over patterns.

Fig. 2. Core syntax

The primary difference between Yhc-Core and GHC-Core [22] is that Yhc-Core is untyped. The Core is generated from well-typed Haskell, and is guaranteed not to fail with a type error. All the transformations could be implemented equally well in a typed Core language, but we prefer to work in an untyped language for simplicity of implementation.

In order to avoid accidental variable name clashes while performing transformations, we demand that all variables within a program are unique. All transformations may assume this invariant, and must ensure it as a postcondition.

3 Optimisation

Our optimisation procedure takes a Core program as input, and produces a new equivalent Core program as output. To improve the program we do not make small local changes to the original, but instead *evaluate it* so far as possible at compile time, leaving a *residual program* to be run.

Each function in the output program is an optimised version of some associated expression in the input program. Optimisation starts at the main function, and optimises the expression associated with main. Once the expression has been optimised, the outermost element in the expression becomes part of the residual program. All the subexpressions are assigned names, and will be given definitions in the residual program. If any expression (up to alpha renaming) already has a name in the residual program, then the same name is used. Each of these named inner expressions is then optimised as before.

Optimisation uses the $\mathcal{O}$ rules in Figure 3, and the simplification rules in Figure 4. We define $\mathcal{O}^*$ to be the result of applying both $\mathcal{O}$ and the simplification rules until no further changes are made. Optimisation is like evaluation, but stops if the expression to reduce is a free variable, a constructor, a primitive, or a CAF (constant applicative form – see §3.3 for more details). The one difference is that in a **let** expression the bound expression and the inner expression are *both*

$$
\begin{array}{lll}
\mathcal{O}[\![\textbf{case}\ x\ \textbf{of}\ \overrightarrow{alts}]\!] = & \textbf{case}\ \mathcal{O}[\![x]\!]\ \textbf{of}\ \overrightarrow{alts} & \\
\mathcal{O}[\![\textbf{let}\ v = x\ \textbf{in}\ y]\!] = & \textbf{let}\ v = \mathcal{O}[\![x]\!]\ \textbf{in}\ \mathcal{O}[\![y]\!] & \\
\mathcal{O}[\![x\ y]\!] = & \mathcal{O}[\![x]\!]\ y & \\
\mathcal{O}[\![f]\!] = & \mathsf{unfold}\ f, \textbf{where}\ f \text{ is a non-primitive, non-CAF function} & \\
\quad = & f\ \ , \mathsf{otherwise} & \\
\mathcal{O}[\![v]\!] = & v & \\
\mathcal{O}[\![c]\!] = & c & \\
\mathcal{O}[\![\lambda v \to x]\!] = & \lambda v \to x &
\end{array}
$$

Fig. 3. Optimisation rules.

optimised – see §3.2 for the reasons. The simplification rules are all standard, and similar rules would be found in most optimising compilers.

Example 1

```
main = λxs → map inc xs

map = λf → λxs → case xs of
                    []     → []
                    y : ys → f y : map f ys

inc = λx → x+1
```

This program defines a main function which increments each value in the list by one. Our main function is not a valid Haskell program, as it has the wrong type, but serves to illustrate the techniques. Note that f is passed around at run-time, when it could be frozen in at compile time. By following the optimisation procedure we end up with:

```
main = λxs → case xs of
               []     → []
               y : ys → f0 y ys

f0 = λy → λys → (y+1) : main ys
```

And finally by performing some trivial inlining we can obtain:

```
main = λxs → case xs of
               []     → []
               y : ys → (y+1) : main ys
```

The residual program is now optimised – there is no runtime passing of the inc function, only a direct arithmetic operation. □

$\textsf{\textbf{case}}\ (\textsf{\textbf{case}}\ x\ \textsf{\textbf{of}}\ \{p_1 \rightarrow y_1; ...; p_n \rightarrow y_n\})\ \textsf{\textbf{of}}\ \overrightarrow{alts}$
$\quad\Rightarrow \textsf{\textbf{case}}\ x\ \textsf{\textbf{of}}\ \{p_1 \rightarrow \textsf{\textbf{case}}\ y_1\ \textsf{\textbf{of}}\ \overrightarrow{alts}$
$\qquad\qquad\qquad ; ...$
$\qquad\qquad\qquad ; p_n \rightarrow \textsf{\textbf{case}}\ y_n\ \textsf{\textbf{of}}\ \overrightarrow{alts}\}$

$\textsf{\textbf{case}}\ c\ \overrightarrow{xs}\ \textsf{\textbf{of}}\ \{...; c\ \overrightarrow{vs} \rightarrow y; ...\}$
$\quad\Rightarrow y\ [\overrightarrow{vs}\ /\ \overrightarrow{xs}]$

$\textsf{\textbf{case}}\ v\ \textsf{\textbf{of}}\ \{...; c\ \overrightarrow{vs} \rightarrow x; ...\}$
$\quad\Rightarrow \textsf{\textbf{case}}\ v\ \textsf{\textbf{of}}\ \{...; c\ \overrightarrow{vs} \rightarrow x\ [v\ /\ c\ \overrightarrow{vs}]; ...\}$

$\textsf{\textbf{case}}\ (\textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ y)\ \textsf{\textbf{of}}\ \overrightarrow{alts}$
$\quad\Rightarrow \textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ \textsf{\textbf{case}}\ y\ \textsf{\textbf{of}}\ \overrightarrow{alts}$

$(\textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ y)\ z$
$\quad\Rightarrow \textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ y\ z$

$(\textsf{\textbf{case}}\ x\ \textsf{\textbf{of}}\ \{p_1 \rightarrow y_1; ...; p_n \rightarrow y_n\})\ z$
$\quad\Rightarrow \textsf{\textbf{case}}\ x\ \textsf{\textbf{of}}\ \{p_1 \rightarrow y_1\ z; ...; p_n \rightarrow y_n\ z\}$

$(\lambda v \rightarrow x)\ y$
$\quad\Rightarrow \textsf{\textbf{let}}\ v = y\ \textsf{\textbf{in}}\ x$

$\textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ (\textsf{\textbf{case}}\ y\ \textsf{\textbf{of}}\ \{p_1 \rightarrow y_1; ...; p_n \rightarrow y_n\})$
$\quad\Rightarrow \textsf{\textbf{case}}\ y\ \textsf{\textbf{of}}\ \{p_1 \rightarrow \textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ y_1$
$\qquad\qquad\qquad ; ...$
$\qquad\qquad\qquad ; p_n \rightarrow \textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ y_n\}$
where v is not used in$y$

$\textsf{\textbf{let}}\ v = x\ \textsf{\textbf{in}}\ y$
$\quad\Rightarrow y\ [v\ /\ x]$
where x is a lambda, variable, or used once in y

$\textsf{\textbf{let}}\ v = c\ x_1 ... x_n\ \textsf{\textbf{in}}\ y$
$\quad\Rightarrow \textsf{\textbf{let}}\ v_1 = x_1\ \textsf{\textbf{in}}$
$\qquad ...$
$\qquad \textsf{\textbf{let}}\ v_n = x_n\ \textsf{\textbf{in}}$
$\qquad y\ [v\ /\ c\ x_1 ... x_n]$
where $v_1 ... v_n$ are fresh

Fig. 4. Simplification rules.

Example 2

Our next example shows how our optimisation rules can carry out list deforestation [23].

```
main xs = map (+1) (map (*2) xs)

map f xs = case xs of
               []     → []
               y : ys → f y : map f ys
```

The main definition is transformed (after trivial inlining) into:

```
main xs = case xs of
              []     → []
              y : ys → (y*2)+1 : main ys
```

The intermediate list has been removed, and the higher order functions eliminated by specialisation. □

3.1 Termination

A problem with the method as presented so far is that it may not terminate. There are several ways that non-termination can arise. We consider, and eliminate, each in turn.

Infinite Unfolding Consider the definition:

```
name = λx → name x
```

If the expression name x was being optimised then the optimisation function $\mathcal{O}^*$ would not terminate. We can solve this problem by either bounding the number of unfoldings, or by keeping a list of previously encountered intermediate expressions in $\mathcal{O}^*$. In practice, this situation is rare, and either choice is acceptable. We choose to bound the number of unfoldings. A large limiting value is used, which does not impact either compilation time or memory consumption in the common case.

Accumulating parameters Consider the definition:

```
reverseAcc = λxs → λys → case xs of
                             []     → []
                             z : zs → reverseAcc zs (z : ys)
```

This function is the standard reverse function, with an accumulator. The problem is that successive iterations of the optimisation produce progressively larger subexpressions. A definition is first created for reverseAcc _ _, then for reverseAcc _ (_ : _), then reverseAcc _ (_ : _ : _). The residual program is infinite.

The solution is to bound the size of the input expression associated with each definition in the residual program. The size of the expression being optimised can be reduced by lifting subexpressions into a let binding, then placing this let binding in the residual program. By bounding the size of the expression, we bound the number of functions in the residual program.

If the bound is too high, optimisation takes too long and the residual program is excessively large. If the bound is too low then too little is achieved by optimisation. We return to the issue of the size of this bound in §5.2.

Direct Repetition We claim that $\mathcal{O}^*$ terminates with bounded unfoldings and bounded expression size. It is often useful to detect an expression which appears to be repeating, and preemptively bound it. Consider the reverseAcc example – the recursive pattern is an instance of *direct repetition*. Let α be a context, and $\alpha\langle e\rangle$ be the result of substituting e for the hole in the context α. An expression x is directly repeating if $x \simeq \alpha\langle\alpha\langle\beta\rangle\rangle$ where β is an expression, α is a non-empty context and $\simeq$ is equality where all variables are considered equal.

Example 3

The following expressions have direct repetition.

$x : y : xs$ **where** $\alpha = x : \bullet, \beta = xs$
$f\ (f\ x)$ **where** $\alpha = f\ \bullet, \beta = x$
case x_1 **of** $\{[\,] \rightarrow \mathsf{nil}; y : ys \rightarrow$ **case** x_2 **of** $\{[\,] \rightarrow \mathsf{nil}; z : zs \rightarrow \mathsf{cons}\}\}$
  where $\alpha =$ **case** x_1 **of** $\{[\,] \rightarrow \mathsf{nil}; y : ys \rightarrow \bullet\}, \beta = \mathsf{cons}$

□

If direct repetition is encountered, then the repeating expression is lifted to a top-level let binding, and output directly into the residual program.

Example 4

Take the reverseAcc example. During optimisation, the expression becomes:

reverseAcc $xs\ (y_1 : y_2 : ys)$

The second argument to reverseAcc is an instance of direct repetition, and is lifted to a let binding.

let $v = y_1 : y_2 : ys$
in reverseAcc $xs\ v$

Now the expression bound at the let, and the inner expression, are optimised separately. □

3.2 Let Bindings

The rule for let bindings in Figure 3 may seem curious. The other rules simply follow evaluation order, but the let rule optimises *both* the bound expression and the inner expression. This is a critical choice, which enhances the optimisation performed by the system, without duplicating computation of let bindings.

In the Core language a let expression introduces a binding, which is shared. Given the expression **let** $v = x$ **in** y, even if v is referred to multiple times in y, then the expression x is computed at most once. It is important that sharing of *expensive* functions is preserved. Yet, by inlining *cheap* let expressions, better optimisation can be achieved. Taking the following fragment from a previous example:

let f = inc
in $f\ y$: map $f\ ys$

If f is not inlined, then the recursive call to map would still contain a functional variable to be passed at runtime. But how can we tell whether inc is cheap enough to be inlined? The solution is to optimise inc first:

let $f = \lambda x \to x{+}1$
in $f\ y$: map $f\ ys$

It is now clear that f is a lambda, so no shared computation is lost by inlining it.

3.3 CAF's

A CAF (constant applicative form) is a top level definition of zero arity. In Haskell, CAFs are computed at most once per program run, and retained as long as references to them remain. Consider the program:

caf = expensive

main = caf+caf

In this program caf would only be computed once. If a CAF is inlined then this may result in a computation being performed more than would otherwise occur. To ensure that we do not duplicate computations, we never inline CAF's.

4 Performance Compared With C Programs

The benchmarks we have chosen are inspired by the Unix `wc` command – namely character, word and line counting. We require the program to read from the standard input, and write out the number of elements in the file. To ensure that we test computation speed, not IO speed (which is usually determined by the buffering strategy, rather than optimisation) we demand that all input is read

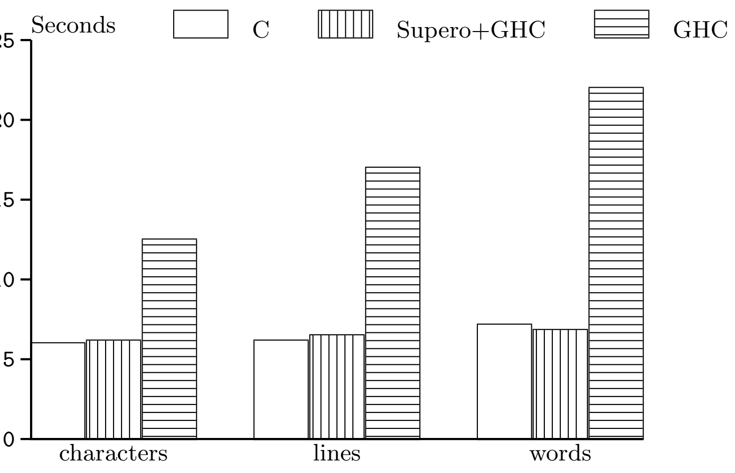

Fig. 5. Benchmarks with C, Supero+GHC and GHC alone.

using the standard C `getchar` function only. Any buffering improvements, such as reading in blocks or memory mapping of files, could be performed equally in all compilers.

All the C versions are implemented following a similar pattern to Figure 1. Characters are read in a loop, with an accumulator recording the current value. Depending on the program, the body of the loop decides when to increment the accumulator. The Haskell versions all follow the same pattern as in the Introduction, merely replacing `words` with `lines`, or removing the `words` function for character counting.

We performed all benchmarks on a machine running Windows XP, with a 3GHz processor and 1Gb RAM. All benchmarks were run over a 50Mb log file, repeated 10 times, and the lowest value was taken. The C versions used GCC[1] version 3.4.2 with -O3. The Haskell version used GHC [21] 6.6.1 with -O2. The Supero version was compiled using our optimiser, then written back as a Haskell file, and compiled once more with GHC 6.6.1 and -O2.

The results are given in Figure 5. In all the benchmarks C and Supero are within 10% of each other, while GHC trails further behind.

4.1 Identified Haskell Speedups

During initial trials using these benchmarks, we identified two unnecessary bottlenecks in the Haskell version of word counting. Both were remedied before the presented results were obtained.

[1] `http://gcc.gnu.org/`

```
words :: String → [String]
words s = case dropWhile isSpace s of
                    [] → []
                    x  → w : words y
                         where (w, y) = break isSpace x

words′ s = case dropWhile isSpace s of
                    []     → []
                    x : xs → (x : w) : words′ (drop1 z)
                              where (w, z) = break isSpace xs

drop1 []       = []
drop1 (x : xs) = xs
```

Fig. 6. The words function from the Haskell standard libraries, and an improved words′.

Slow isSpace function The first issue is that isSpace in Haskell is much more expensive than isspace in C. The simplest solution is to use a FFI (Foreign Function Interface) [14] call to the C isspace function in all cases, removing this factor from the benchmark. A GHC bug (number 1473) has been filed about the slow performance of isSpace.

Inefficient words function The second issue is that the standard definition of the words function (given in Figure 6) performs two additional isSpace tests per word. By appealing to the definitions of dropWhile and break it is possible to show that in words the first character of x is not a space, and that if y is non-empty then the first character is a space. The revised words′ function uses these facts to avoid the redundant isSpace tests.

4.2 Potential GHC Speedups

We have identified three factors limiting the performance of residual programs when compiled by GHC. These problems cannot be solved at the level of Core transformations. We suspect that by fixing these problems, the Supero execution time would improve by between 5% and 15%.

Strictness inference The GHC compiler is overly conservative when determining strictness for functions which use the FFI (GHC bug 1592). The getchar function is treated as though it may raise an exception, and terminate the program, so strict arguments are not determined to be strict. If GHC provided some way to mark an FFI function as not generating exceptions, this problem could be solved. The lack of strictness information means that in the line and word counting programs, every time the accumulator is incremented, the number is first unboxed and then reboxed [17].

Heap checks The GHC compiler follows the standard STG machine [12] design, and inserts heap checks before allocating memory. The purpose of a heap check is to ensure that there is sufficient memory on the heap, so that allocation of memory is a cheap operation guaranteed to succeed. GHC also attempts to lift heap checks: if two branches of a case expression both have heap checks, they are replaced with one shared heap check before the case expression. Unfortunately, with lifted heap checks, a tail-recursive function that allocates memory only upon exit can have the heap test executed on every iteration (GHC bug 1498). This problem affects the character counting example, but if the strictness problems were solved, it would apply equally to all the benchmarks.

Stack checks The final source of extra computation relative to the C version are stack checks. Before using the stack to store arguments to a function call, a test is performed to check that there is sufficient space on the stack. Unlike the heap checks, it is necessary to analyse a large part of the flow of control to determine when these checks are unnecessary. So it is not clear how to reduce stack checks in GHC.

4.3 Why Supero Outperforms C for the Wordcount Benchmark

The most curious result is that Supero outperforms C on wordcounting, by about 6% – even with the problems discussed! The C program presented in Figure 1 is not optimal. The variable `last_space` is a boolean, indicating whether the previous character was a space, or not. Each time round the loop a test is performed on `last_space`, even though its value was determined and tested on the previous iteration. The way to optimise this code is to have two specialised variants of the loop, one for when `last_space` is true, and one for when it is false. When the value of `last_space` changes, the program would transition to the other loop. This pattern effectively encodes the boolean variable in the program counter, and is what the Haskell program has managed to generate from the high-level code.

However, in C it is quite challenging to capture the required control flow! The program needs two loops, where both loops can transition to the other. Using `goto` turn off many critical optimisations in the C compiler. Tail recursion is neither required by the C standard, nor supported by most compilers. The only way to express the necessary pattern is using nested while loops, but unlike newer imperative languages such as Java, C does not have named loops – so the inner loop cannot break from the outer loop if it reaches the end of the file. The only solution is to place the nested while loops in a function, and use `return` to break from the inner loop. This solution would not scale to a three-valued control structure, and substantially increases the complexity of the code.

5 Performance Compared With GHC Alone

The standard set of Haskell benchmarks is the nofib suite [11]. It is divided into three categories of increasing size: imaginary, spectral and real. Many small

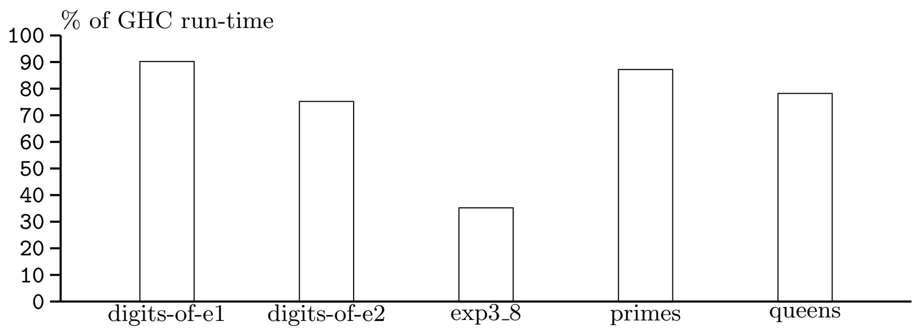

Fig. 7. Runtime, relative to GHC.

Program	**Source**	**Residual**	**Bound**	**% GHC Size**	**% GHC Time**
digits-of-e1	521	1676	13	110	90
digits-of-e2	1235	515	12	99	75
exp3_8	380	1138	5	104	35
primes	422	356	12	101	87
queens	637	4265	8	116	78

Program is the name of the program; **Source** is the number of lines of pretty printed source code including all libraries; **Residual** is the number of lines after optimisation; **Bound** is the termination bound used; **Size** is the size of the resultant binary as a percentage of the GHC binary size; **Time** is the runtime as a percentage of GHC run-time.

Table 1. Result on the nofib suite.

Haskell programs increase in size substantially once the libraries are included, particularly when type classes are involved. Because of the relatively large source code size of even small examples, we have limited our focus to five benchmarks drawn from the imaginary section. We have chosen programs which do not perform large amounts of IO.

The benchmarks are: digits-of-e1 and digits-of-e2, both of which compute the digits of e by different methods; exp3_8 computes 3^8 using Peano numbers and the Num class; primes computes a list of prime numbers; and queens counts the safe layouts of queen pieces on a chess board. All benchmarks were run with parameters that require runtimes of between 3 and 5 seconds for GHC.

The results of these benchmarks are given in Figure 7, along with detailed breakdowns in Table 1. In all benchmarks Supero+GHC performs at least 10% faster than GHC alone, and in one case is nearly three times faster. Binaries were at most 10% larger than those from GHC alone, and in one case the binary was even marginally smaller.

5.1 GHC's optimisations

For these benchmarks it is important to clarify which optimisations are performed by GHC, and which are performed by Supero. Core output from Yhc, compiled using GHC without any prior optimisation, would *not* perform as well as the original program compiled using GHC. GHC has two special optimisations that work in a restricted number of cases, but which Supero is unable to take advantage of.

Dictionary Removal Functions which make use of type classes are given an additional dictionary argument. In practice, GHC specialises many such functions by creating code with a particular dictionary frozen in. This optimisation is specific to type classes – a tuple of higher order functions is not similarly specialised. After compilation with Yhc, the type classes have already been converted to tuples, so Supero must be able to remove the dictionaries itself. One benchmark where dictionary removal is critical is digits-of-e2.

List Fusion GHC relies on names of functions, particularly `foldr/build` [19], to apply special optimisation rules such as list fusion. Many of GHC's library functions, for example `iterate`, are defined in terms of `foldr` to take advantage of these special properties. After transformation with Yhc, these names are destroyed, so no rule based optimisation can be performed. One example where list fusion is critical is primes, although it occurs in most of the benchmarks to some extent.

Supero has no special purpose optimisations which rely on named functions or desugaring knowledge. The one benchmark where no GHC specific optimisations apply is exp3_8, which operates solely on Peano numbers – a type GHC has no inbuilt knowledge of. Hence the advantage of Supero in exp3_8: while GHC is limited to basic inline/simplify transformations, Supero is able to remove some intermediate data structures.

5.2 Termination Bound

Table 1 includes a column indicating the size bound that was applied to expressions. Out of the five benchmarks, both primes and queens could be run at any greater bound and would still produce the same program – the direct repetition criteria (see §3.1) bounds the expressions on its own. For the remaining programs, a bound was chosen to ensure that the compilation process was quick (under two seconds). By increasing the termination bound the size of the residual program would increase, but the generated program may execute faster.

The existence of a termination bound requiring different values for different programs is a cause for concern. In a large program it is likely that different parts of the program would require different bounds on the size of the generated expression – something not currently possible. We suspect that the most promising direction is to augment the direct repetition criterion to obtain termination in all practical cases without resorting to a depth bound.

6 Related Work

Partial evaluation There has been a lot of work on partial evaluation [7], where a program is specialised with respect to some static data. The emphasis is on determining which variable can be entirely computed at compile time, and which must remain in the residual program. Partial evaluation is particularly appropriate for specialising an interpreter with an expression tree to generate a compiler automatically, often with an order of magnitude speedup, known as the First Futamura Projection [4]. The difference between our work and partial evaluation is that we fold back definitions, and perform no binding time analysis. Our method is certainly less appropriate for specialising an interpreter, but in the absence of static data, is still able to show improvements.

Deforestation The deforestation technique [23] removes intermediate lists in computations. This technique has been extended in many ways to encompass higher order deforestation [8] and work on other data types [3]. Probably the most practically motivated work on deforestation has come from those attempting to restrict deforestation, in particular shortcut deforestation [5], and newer approaches such as stream fusion [2]. In this work certain named functions are automatically fused together. By rewriting library functions in terms of these special functions, fusion occurs. Shortcut deforestation is limited to cases where the correct underlying function is used – sometimes requiring unnatural definitions.

GRIN The GRIN approach [1] is currently being implemented in the jhc compiler [10], with promising initial results. GRIN works by first translating to a monadic intermediate language, then repeatedly performing a series of optimisations, using whole program transformation. The intermediate language is at a much lower level than our Core language, so jhc is able to perform detailed optimisations that we are unable to express.

Other Transformations One of the central operations within our optimisation in inlining, a technique that has been used extensively within GHC [18]. We generalise the constructor specialisation technique [16], by allowing specialisation on any arbitrary expression, including constructors.

Lower Level Optimisations Our optimisation works at the Core level, but even once optimal Core has been generated there is still some work before optimal machine code can be produced. Key optimisations include strictness analysis and unboxing [17]. In GHC both of these optimisations are done at the Core level, using a Core language extended with unboxed types. After this lower level Core has been generated, it is then transformed in to STG machine instructions [13], before being transformed into assembly code. There is still work being done to modify the lowest levels to take advantage of the current generation of microprocessors [9]. We rely on GHC to perform all these optimisations after Supero generates a residual program.

7 Conclusions and Future Work

We have introduced an optimising front-end which can enhance the results of back-end compilation using GHC – at least for some small programs. Our optimiser is simple – the Core transformation is expressed in just 300 lines of Haskell, yet it replicates many of the performance enhancements of GHC in a more general way. Our initial results are promising, but incomplete. There are three main obstacles that need to be tackled:

Termination We are confident that Supero terminates, but only by use of a crude bound on expression size whose optimal value varies for different programs. To increase the applicability of our optimiser, we would like to remove the depth bound, or at least reduce our reliance upon it.

Benchmarks Eight small benchmarks are not enough. We would like to obtain results for all the remaining benchmarks in the nofib suite.

Performance The performance results presented in §5 are disappointing. Earlier versions of Supero were able to obtain a 50% speed up in the primes benchmark, but decreased performance in other benchmarks. We suspect that much better performance can be obtained.

The Programming Language Shootout[2] has shown that low-level Haskell can compete with with low-level imperative languages such as C. Our goal is that Haskell programs can be written in a high-level declarative style, yet still perform competitively.

Acknowledgements We would like to thank Simon Peyton Jones, Simon Marlow and Tim Chevalier for help understanding the low-level details of GHC, and Peter Jonsson for helpful discussions.

References

1. Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proc IFL '96*, volume 1268 of *LNCS*, pages 58–84. Springer-Verlag, 1996.
2. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc ICFP '07*. ACM Press, April 2007.
3. Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Proc PADL 2007*, pages 50–64. Springer-Verlag, January 2007.
4. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
5. Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proc FPCA '93*, pages 223–232. ACM Press, June 1993.
6. Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.
7. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[2] `http://shootout.alioth.debian.org/`

8. Simon Marlow. *Deforestation for Higher-Order Functional Programs.* PhD thesis, University of Glasgow, 1996.
9. Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proc. ICFP '07*. ACM Press, October 2007.
10. John Meacham. jhc: John's haskell compiler. `http://repetae.net/john/computer/jhc/`, 2007.
11. Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2007.
12. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
13. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *JFP*, 2(2):127–202, 1992.
14. Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction, Marktoberdorf Summer School*, 2002.
15. Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
16. Simon Peyton Jones. Constructor specialisation for Haskell programs. In *Proc. ICFP '07*. ACM Press, October 2007.
17. Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proc FPCA '91*, volume 523 of *LNCS*, pages 636–666, Cambridge, Massachussets, USA, August 1991. Springer-Verlag.
18. Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12:393–434, July 2002.
19. Simon Peyton-Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proc. Haskell '01*, pages 203–233. ACM Press, 2001.
20. Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In *Proc. FPCA '95*, pages 282–292. ACM Press, 1995.
21. The GHC Team. The GHC compiler, version 6.6. `http://www.haskell.org/ghc/`, October 2006.
22. Andrew Tolmach. An External Representation for the GHC Core Language. `http://www.haskell.org/ghc/docs/papers/core.ps.gz`, September 2001.
23. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc ESOP '88*, volume 300 of *LNCS*, pages 344–358. Berlin: Springer-Verlag, 1988.
24. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.