A Supercompiler for Core Haskell

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 present many challenges for optimising compilers. We report practical experiments using novel variants of *supercompilation*, with special attention to let bindings and the generalisation technique.

1 Introduction

Haskell [17] 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:

$$\mathsf{main} = \mathsf{print} \circ \mathsf{length} \circ \mathsf{words} =\!\ll \mathsf{getContents}$$

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 [25]) 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, named Supero, we can eliminate all these overheads. We obtain a program that performs *faster* than the C version. The optimiser is based around the techniques of supercompilation [29], where some of the program is evaluated at compile time, leaving an optimised residual program.

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 Contributions

- To our knowledge, this is the first time supercompilation has been applied to Haskell.
- We make careful study of the let expression, something absent from the Core language of many other papers on supercompilation.
- We present an alternative generalisation step, based on a homeomorphic embedding [9].

1.2 Roadmap

We first introduce a Core language in §2, on which all transformations are applied. Next we describe our supercompilation 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.

$$
\begin{array}{lll}
\mathsf{expr} = v & & \text{variable} \\
\quad \mid c & & \text{constructor} \\
\quad \mid f & & \text{function} \\
\quad \mid x\ \overline{ys} & & \text{application} \\
\quad \mid \lambda\overline{vs} \to x & & \text{lambda abstraction} \\
\quad \mid \textbf{let}\ v = x\ \textbf{in}\ y & & \text{let binding} \\
\quad \mid \textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\} & & \text{case expression} \\
\\
\mathsf{pat}\ = c\ \overline{vs} & &
\end{array}
$$

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

Fig. 2. Core syntax

$$
\begin{array}{rl}
\mathsf{split}\ (v) & = (v, [\,]) \\
\mathsf{split}\ (c) & = (c, [\,]) \\
\mathsf{split}\ (f) & = (f, [\,]) \\
\mathsf{split}\ (x\ \overline{ys}) & = (\bullet\ \overline{\bullet}, x : \overline{ys}) \\
\mathsf{split}\ (\lambda\overline{vs} \to x) & = (\lambda\overline{vs} \to \bullet, x) \\
\mathsf{split}\ (\textbf{let}\ v = x\ \textbf{in}\ y) & = (\textbf{let}\ v = \bullet\ \textbf{in}\ \bullet\,, [x, y]) \\
\mathsf{split}\ (\textbf{case}\ x\ \textbf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\}) & = (\textbf{case}\ \bullet\ \textbf{of}\ \{p_1 \to \bullet; ...; p_n \to \bullet\}, [x, y_1, ..., y_n])
\end{array}
$$

Fig. 3. The split function, returning a spine and all subexpressions.

2 Core Language

Our supercompiler uses the Yhc-Core language [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 using the dictionary transformation [32].

The Yhc compiler, a fork of nhc [22], can output Core files. Yhc can also link in all definitions from all required libraries, producing a single Core file representing a whole program.

The primary difference between Yhc-Core and GHC-Core [26] 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 maintain it.

```
supercompile ()
  seen := { }
  bind := { }
  tie ({ }, main)

tie (ρ, x)
  if x ∉ seen then
    seen := seen ∪ { x }
    bind := bind ∪ { ψ(x) = λfv(x) → drive(ρ, x) }
  endif
  return (ψ(x) fv(x))

drive (ρ, x)
  if terminate(ρ, x) then
    (a, b) = split(generalise(x))
    return join(a, map (tie ρ) b)
  else
    return drive(ρ ∪ { x }, unfold (x))
```

Where ψ is a mapping from expressions to function names, and fv(x) returns the free variables in x. This code is parameterised by: terminate which decides whether to stop supercompilation of this expression; generalise which generalises an expression before residuation; unfold which chooses a function application and unfolds it.

Fig. 4. The supercompile function.

We define the split function in Figure 3, which splits an expression into a pair of its spine and its immediate subexpressions. The • markers in the spine indicate the positions from which subexpressions have been removed. We define the join operation to be the inverse of split, taking a spine and a list of expressions, and producing an expression.

3 Supercompilation

Our supercompiler takes a Core program as input, and produces an 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.

The general method of supercompilation is shown in Figure 4. Each function in the output program is an optimised version of some associated expression in the input program. Supercompilation starts at the main function, and supercompiles the expression associated with main. Once the expression has been supercompiled, 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 are then supercompiled as before.

The supercompilation of an expression proceeds by repeatedly inlining a function application until some termination criterion is met. Once the termination criterion holds, the expression is generalised before the outer spine becomes part of the residual program and all immediate subexpressions are assigned names.

$\mathsf{case}\ (\mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})\ \mathsf{of}\ \overline{alts}$
$\quad \Rightarrow \mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to \mathsf{case}\ y_1\ \mathsf{of}\ \overline{alts}$
$\qquad\qquad\qquad ; ...$
$\qquad\qquad\qquad ; p_n \to \mathsf{case}\ y_n\ \mathsf{of}\ \overline{alts}\}$

$\mathsf{case}\ c\ x_1 ... x_n\ \mathsf{of}\ \{...; c\ v_1 ... v_n \to y; ...\}$
$\quad \Rightarrow \mathsf{let}\ v_1 = x_1\ \mathsf{in}$
$\qquad ...$
$\qquad \mathsf{let}\ v_n = x_n\ \mathsf{in}$
$\qquad y$

$\mathsf{case}\ v\ \mathsf{of}\ \{...; c\ \overline{vs} \to x; ...\}$
$\quad \Rightarrow \mathsf{case}\ v\ \mathsf{of}\ \{...; c\ \overline{vs} \to x\ [v\ /\ c\ \overline{vs}]; ...\}$

$\mathsf{case}\ (\mathsf{let}\ v = x\ \mathsf{in}\ y)\ \mathsf{of}\ \overline{alts}$
$\quad \Rightarrow \mathsf{let}\ v = x\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \overline{alts}$

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

$(\mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to y_1; ...; p_n \to y_n\})\ z$
$\quad \Rightarrow \mathsf{case}\ x\ \mathsf{of}\ \{p_1 \to y_1\ z; ...; p_n \to y_n\ z\}$

$(\lambda v \to x)\ y$
$\quad \Rightarrow \mathsf{let}\ v = y\ \mathsf{in}\ x$

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

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

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

Fig. 5. Simplification rules.

After each inlining step, the expression is simplified using the rules in Figure 5. There are three key decisions in the supercompilation of an expression:

1. Which function to inline.
2. What termination criterion to use.
3. What generalisation to use.

The original Supero work [13] inlined following evaluation order (with the exception of let expressions), used a bound on the size of the expression to ensure termination, and performed no generalisation. First we give examples of our supercompiler in use, then we return to examine each of the three choices we have made.

3.1 Examples of Supercompilation

Example 1: Supercompiling and Specialisation

$$\mathsf{main}\ as = \mathsf{map}\ (\lambda b \to b{+}1)\ as$$

$$\begin{array}{l}\mathsf{map}\ f\ cs = \textbf{case}\ cs\ \textbf{of} \\ \qquad\qquad [\,] \quad\ \to [\,] \\ \qquad\qquad d : ds \to f\ d : \mathsf{map}\ f\ ds\end{array}$$

There are two primary inefficiencies in this example: (1) the map function passes the f argument invariantly in every call; (2) the application of f is more expensive than if the function was known in advance.

The supercompilation proceeds by first assigning a new unique name (we choose h_0) to $\mathsf{map}\ (\lambda b \to b{+}1)\ as$, providing parameters for each of the free variables in the expression, namely as. We then choose to expand map, and invoke the simplification rules:

$$\begin{array}{l}h_0\ as = \mathsf{map}\ (\lambda b \to b{+}1)\ as \\ \\ \quad = \textbf{case}\ as\ \textbf{of} \\ \qquad\quad [\,] \quad\ \to [\,] \\ \qquad\quad d : ds \to d{+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ ds\end{array}$$

We now have a **case** with a variable as the scrutinee at the root of the expression, which cannot be reduced further, so we residuate the spine. When processing the expression $\mathsf{map}\ (\lambda b \to b{+}1)\ ds$ we spot this to be an alpha renaming of the body of an existing generated function, namely h_0, and use this function:

$$\begin{array}{l}h_0\ as = \textbf{case}\ as\ \textbf{of} \\ \qquad\qquad [\,] \quad\ \to [\,] \\ \qquad\qquad d : ds \to d{+}1 : h_0\ ds\end{array}$$

We have now specialised the higher-order argument, passing less data at runtime. □

Example 2: Supercompiling and Deforestation

The deforestation transformation [31] removes intermediate lists from a traversal. A similar result is obtained by applying supercompilation, as shown here. Consider the operation of mapping $(*2)$ over a list and then mapping $(+1)$ over the result. The first map deconstructs one list, and constructs another. The second does the same.

$$\mathsf{main}\ as = \mathsf{map}\ (\lambda b \to b{+}1)\ (\mathsf{map}\ (\lambda c \to c{*}2)\ as)$$

We first assign a new name for the body of main, then choose to expand the outer call to map:

$$\begin{array}{l} h_0\ as = \textbf{case}\ \mathsf{map}\ (\lambda c \to c{*}2)\ as\ \textbf{of} \\ \qquad\qquad [\,] \quad \to [\,] \\ \qquad\qquad d : ds \to d{+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ ds \end{array}$$

Next we choose to inline the map scrutinised by the case, then perform the **case**/**case** simplification, and finally residuate:

$$\begin{array}{l} h_0\ as = \textbf{case}\ (\textbf{case}\ as\ \textbf{of} \\ \qquad\qquad\qquad [\,] \quad \to [\,] \\ \qquad\qquad\qquad e : es \to e{*}2 : \mathsf{map}\ (\lambda c \to c{*}2)\ es)\ \textbf{of} \\ \qquad\qquad [\,] \quad \to [\,] \\ \qquad\qquad d : ds \to y{+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ ds \\ \\ \quad = \textbf{case}\ as\ \textbf{of} \\ \qquad\qquad [\,] \quad \to [\,] \\ \qquad\qquad d : ds \to (y{*}2){+}1 : \mathsf{map}\ (\lambda b \to b{+}1)\ (\mathsf{map}\ (\lambda c \to c{*}2)\ ds) \\ \\ \quad = \textbf{case}\ as\ \textbf{of} \\ \qquad\qquad [\,] \quad \to [\,] \\ \qquad\qquad d : ds \to (y{*}2){+}1 : h_0\ ds \end{array}$$

Both intermediate lists have been removed, and the functional arguments to map have both been specialised. □

3.2 Which function to inline

During the supercompilation of an expression, at each step some function needs to be inlined. Which to choose? In most supercompilation work the choice is made following the runtime semantics of the program. But in a language with let expressions this may be inappropriate. If a function in a let binding is inlined, its application when reduced may be simple enough to substitute in the let body. However, if a function in a let body is inlined, the let body may now only refer to the let binding once, allowing the binding to be substituted. Let us take two expressions, based on intermediate steps obtained from real programs (word counting and prime number calculation respectively):

```
let x = (≡) $ 1                    let x = repeat 1
in x 1 : map x ys                  in const 0 x : map f x
```

In the first example, inlining ($) in the let binding gives $(\lambda x \to 1 \equiv x)$, which is now simple enough to substitute for x, resulting in $(1 \equiv 1 : \mathsf{map}\ (\lambda x \to 1 \equiv x)\ ys)$ after simplification. Now map can be specialised appropriately. Alternatively, expanding the map repeatedly would keep increasing the size of expression until the termination criterion was met, aborting the supercompilation of this expression without achieving specialisation.

Taking the second example, repeat can be inlined indefinitely. However, by unfolding the const we produce $\mathbf{let}\ x = \mathsf{repeat}\ 1\ \mathbf{in}\ 0 : \mathsf{map}\ f\ x$. Since x is only used once we substitute it to produce $(0 : \mathsf{map}\ f\ (\mathsf{repeat}\ 1))$, which can be deforested.

Unfortunately these two examples seem to suggest different strategies for unfolding – unfold in the let binding or unfold in the let body. However, they do have a common theme – unfold the function that cannot be unfolded infinitely often. Our strategy can be defined by the unfold function:

```
unfold x = head (filter (not ∘ terminate) xs ++ xs ++ [x])
  where xs = unfolds x

unfolds f | f is a function = [inline f]
unfolds x = [join spine (sub |+ (i, y))
             | let (spine, sub) = split x
             , i ← [0 .. length sub], y ← unfolds (sub !! i)]
  where xs |+ (i, x) = zipWith (λj y → if i ≡ j then x else y) [0 ..] xs
```

The unfolds function computes all possible one-step inlinings, using an in-order traversal of the abstract syntax tree. The unfold function chooses the first unfolding which does not cause the supercompilation to terminate. If no such expression exists, the first unfolding is chosen.

3.3 The Termination Criterion

The original Supero program used a size bound on the expression to determine when to stop. The problem with a size bound is that different programs require different bounds to ensure both timely completion at compile-time and efficient residual programs. Indeed, within a single program, there may be different elements requiring different size bounds – a problem exacerbated as the size and complexity of a program increases.

We use the termination criterion suggested by Sørensen and Glück [24] – homeomorphic embedding. An expression x is an embedding of y, written $x \trianglelefteq y$, if the relationship can be inferred by the rules:

$$\frac{\text{dive}(x, y)}{x \trianglelefteq y} \qquad \frac{\text{couple}(x, y)}{x \trianglelefteq y}$$

$$\frac{s \trianglelefteq t_i \text{ for some } i}{\text{dive}(s, \sigma(t_1, \ldots, t_n))} \qquad \frac{\sigma_1 \sim \sigma_2, s_1 \trianglelefteq t_1, \ldots, s_n \trianglelefteq t_n}{\text{couple}(\sigma_1(s_1, \ldots, s_n), \sigma_2(t_1, \ldots, t_n))}$$

The homeomorphic embedding uses the relations dive and couple. The dive relation checks if the first term is contained as a child of the second term, while the couple relation checks if both terms have the same outer shell. We use σ to denote the spine of an expression, with $s_1, \ldots, s_n$ being its subexpressions. We test for equivalence of σ_1 and σ_2 using the $\sim$ relation, a weakened form of equality where all variables are considered equal. We terminate the supercompilation of an expression y if on the chain of reductions from main to y we have encountered an expression x such that $x \trianglelefteq y$.

In addition to using the homeomorphic embedding, we also terminate if further unfolding cannot yield any improvement to the root of the expression. For example, if the root of an expression is a constructor application, no further unfolding will change the root constructor. When terminating for this reason, we always residuate the outer spine of the expression, without applying any generalisation.

3.4 Generalisation

When the termination criterion has been met, it is necessary to discard information about the current expression, so that the supercompilation terminates. We always residuate the outer spine of the expression, but first we attempt to generalise the expression so that the information lost is minimal. The paper by Sørensen and Glück provides a method for generalisation, which works by taking the most specific generalisation of the current expression and expression which is a homeomorphic embedding of it.

The most specific generalisation of two expressions s and t, $\text{msg}(s, t)$, is produced by applying the following rewrite rule to the initial triple $(x, \{x = s\}, \{x = t\})$, resulting in a common expression and two sets of bindings.

$$\begin{pmatrix} t_g \\ \{x = \sigma(s_1, \ldots, s_n)\} \cup \theta_1 \\ \{x = \sigma(t_1, \ldots, t_n)\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g[x/\sigma(y_1, \ldots, y_n)] \\ \{y_1 = s_1, \ldots, y_n = s_n\} \cup \theta_1 \\ \{y_1 = t_1, \ldots, y_n = t_n\} \cup \theta_2 \end{pmatrix}$$

Our generalisation is characterised by $x \bowtie y$, which produces an expression equivalent to y, but similar in structure to x.

$x \bowtie \sigma^*(y)$, if $\text{dive}(x, \sigma^*(y)) \wedge \text{couple}(x, y)$
  let $f = \lambda\overline{vs} \rightarrow x$ **in** $\sigma^*(f\ vs)$
  where $\overline{vs} = \text{fv}(y) \backslash \text{fv}(\sigma^*(y))$

$x \bowtie y$, if $\text{couple}(x, y)$
  let θ_2 **in** t_g
  where $(t_g, \theta_1, \theta_2) = \text{msg}(x, y)$

The fv function in the first rule calculates the free variables of an expression, and $\sigma^*(y)$ denotes a subexpression y within a containing context σ^*. The first rule applies if the homeomorphic embedding first applied the dive rule. The idea is to descend to the element which matched, and then promote this to the top-level using a lambda. The second rule applies the most specific generalisation operation if the coupling rule was applied first. We now show an example where most specific generalisation fails to produce the ideal generalised version.

Example 3

```
case putStr (repeat '1') r of
    (r, _) → (r, ())
```

This expression (which we name x) prints an infinite stream of 1's. The pairs and r's correspond to the implementation of GHC's IO Monad [16]. After several unrollings, we obtain the expression (named x'):

```
case putChar '1' r of
    (r, _) → case putStr (repeat '1') r of
                 (r, _) → (r, ())
```

The homeomorphic embedding $x \trianglelefteq x'$ matches, detecting an occurrence of the **case** putStr ... expression, and the supercompilation of x' is stopped. The most specific generalisation rule is applied as $\text{msg}(x, x')$ and produces:

```
let a = putChar
    b = '1'
    c = λr → case putStr (repeat '1') r of
                 (r, _) → (r, ())
in case a b r of
       (r, _) → c r
```

The problem is that msg works from the top, looking for a common root of both expression trees. However, if the first rule applied by $\trianglelefteq$ was dive, the roots may be unrelated. Using our generalisation, $x \bowtie x'$:

```
let x = λr → case putStr (repeat '1') r of
                 (r, _) → (r, ())
in case putChar '1' r of
       (r, _) → x r
```

Our generalisation is superior because it has split out the putStr application *without* lifting the putChar application or the constant '1'. The putChar application can now be supercompiled further in the context of the case expression. □

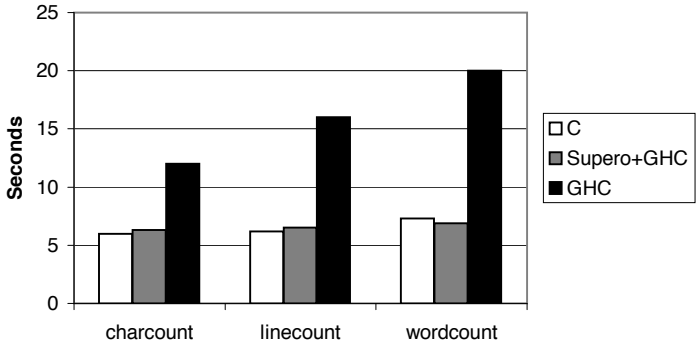


Fig. 6. Benchmarks with C, Supero+GHC and GHC alone.

4 Performance Compared With C Programs

The benchmarks we have used as motivating examples 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 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 6.8.1 with -O2. The Supero version was compiled using our optimiser, then written back as a Haskell file, and compiled once more with GHC 6.8.1 and -O2.

The results are given in Figure 6. In all the benchmarks C and Supero are within 10% of each other, while GHC trails further behind.

[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. 7. The words function from the Haskell standard libraries, and an improved words′.

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.

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) [16] 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 7) 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 [19].

Heap checks The GHC compiler follows the standard STG machine [15] 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.

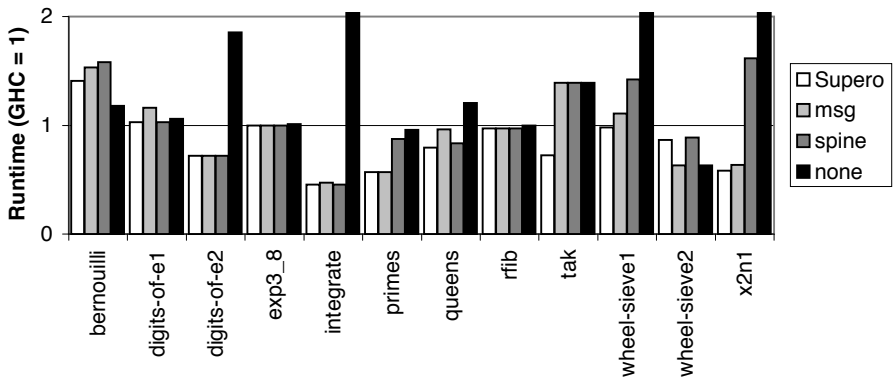
Stack checks The final source of extra computation relative to the C version are stack checks. Before using the stack to store arguments to a function call, a test is performed to check that there is sufficient space on the stack. Unlike the heap checks, it is necessary to analyse a large part of the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

5 Performance Compared With GHC Alone

The standard set of Haskell benchmarks is the nofib suite [14]. It is divided into three categories of increasing size: imaginary, spectral and real. Even small Haskell programs increase in size substantially once libraries are included, so we have limited our attention to the benchmarks in the imaginary section. All benchmarks were run with parameters that require runtimes of between 3 and 5 seconds for GHC.

We exclude two benchmarks, paraffins and gen_regexps. The paraffins benchmark makes substantial use of arrays, and we have not yet mapped the array primitives of Yhc onto those of GHC, which is necessary to run the transformed result. The gen_regexps benchmark tests character processing: for some reason (as yet unknown) the supercompiled executable fails.

The results of these benchmarks are given in Figure 8, along with detailed breakdowns in Table 1. All results are relative to the runtime of a program compiled with GHC -O2, lower numbers being better. The first three variants (Supero, msg, spine) all use homeomorphic embedding as the termination criterion, and $\bowtie$, msg or nothing respectively as the generalisation function. The final variant, none, uses a termination test that always causes a residuation. The 'none' variant is useful as a control to determine which improvements are due to bringing all definitions into one module scope, and which are a result of supercompilation. Compilation times ranged from a few seconds to five minutes.



Supero uses the ⋈ generalisation method; **msg** uses the msg function for generalisation; **spine** applies no generalisation operation; **none** never performs any inlining.

Fig. 8. Runtime, relative to GHC being 1.

The Bernouilli benchmark is the only one where Supero is slower than GHC by more than 3%. The reason for this anomaly is that a dictionary is referred to in an inner loop which is specialised away by GHC, but not by Supero.

With the exception of the wheel-sieve2 benchmark, our ⋈ generalisation strategy performs as well as, or better than, the alternatives. While the msg generalisation performs better than the empty generalisation on average, the difference is not as dramatic.

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. The 'none' results show that, on average, taking the Core output from Yhc and compiling with GHC does *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 produced Core 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 [21], 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

Program	Supero	msg	spine	none	Size	Memory
bernouilli	1.41	1.53	1.58	1.18	1.10	0.97
digits-of-e1	1.03	1.16	1.03	1.06	1.01	1.11
digits-of-e2	0.72	0.72	0.72	1.86	1.00	0.84
exp3_8	1.00	1.00	1.00	1.01	0.99	1.00
integrate	0.46	0.47	0.46	4.01	1.02	0.08
primes	0.57	0.57	0.88	0.96	1.00	0.98
queens	0.79	0.96	0.83	1.21	1.01	0.85
rfib	0.97	0.97	0.97	1.00	1.00	1.08
tak	0.72	1.39	1.39	1.39	1.00	1.00
wheel-sieve1	0.98	1.11	1.42	5.23	1.19	2.79
wheel-sieve2	0.87	0.63	0.89	0.63	1.49	2.30
x2n1	0.58	0.64	1.61	3.04	1.09	0.33

Program is the name of the program; **Supero** uses the $\bowtie$ generalisation method; **msg** uses the msg function for generalisation; **spine** applies no generalisation operation; **none** never performs any inlining; **Size** is the size of the Supero generated executable; **Memory** is the amount of memory allocated on the heap by the Supero executable.

Table 1. Runtime, relative to GHC being 1.

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.

6 Related Work

Supercompilation [29, 30] was introduced by Turchin for the Refal language [28]. Since this original work, there have been various suggestions of both termination strategies and generalisation strategies [27, 24, 9]. The original supercompiler maintained both positive and negative knowledge, but our implementation is a simplified version maintaining only positive information [23].

The issue of let expressions in supercompilation has not previously been a primary focus. If lets are mentioned, the usual strategy is to substitute all linear lets and residuate all others. Lets have been considered in a strict setting [8], where they are used to preserve termination semantics, but in this work all strict lets are inlined without regard to loss of sharing. Movement of lets can have a dramatic impact on performance: carefully designed let-shifting transformations give an average speedup of 15% in GHC [20], suggesting let expressions are critical to the performance of real programs.

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 variables 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]. Partial evaluation is not usually able to remove intermediate data structures. 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 [31] removes intermediate lists in computations. This technique has been extended in many ways to encompass higher order deforestation [10] and work on other data types [3]. Probably the most practically motivated work 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.

Whole Program Compilation The GRIN approach [1] uses whole program compilation for Haskell. It is currently being implemented in the jhc compiler [12], with promising initial results. GRIN works by first removing all functional values, turning them into case expressions, allowing subsequent optimisations. The intermediate language for jhc is at a much lower level than our Core language, so jhc is able to perform detailed optimisations that we are unable to express.

Lower Level Optimisations Our optimisation works at the Core level, but even once efficient Core has been generated there is still some work before efficient machine code can be produced. Key optimisations include strictness analysis and unboxing [19]. 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 compiled to STG machine instructions [15], from which assembly code is generated. There is still work being done to modify the lowest levels to take advantage of the current generation of microprocessors [11]. We rely on GHC to perform all these optimisations after Supero generates a residual program.

7 Conclusions and Future Work

Our supercompiler 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. We have modified some of the techniques from supercompilation, particularly with respect to let bindings and generalisation. Our initial results are promising, but incomplete. Using our supercompiler in conjunction with GHC we obtain an average runtime improvement of 16% for the imaginary section of the nofib suite. To quote Simon Peyton Jones, "an average runtime improvement of 10%, against the baseline of an already well-optimised compiler, is an excellent result" [18].

There are three main areas for future work:

More Benchmarks The fifteen benchmarks presented in this paper are not enough. We would like to obtain results for larger programs, including all the remaining benchmarks in the nofib suite.

Runtime Performance Earlier versions of Supero [13] managed to obtain substantial speed ups on benchmarks such as exp3_8. The Bernouilli benchmark is currently problematic. There is still scope for improvement.

Compilation Speed The compilation times are tolerable for benchmarking and a final optimised release, but not for general use. Basic profiling shows that over 90% of supercompilation time is spent testing for a homeomorphic embedding, which is currently done in a naïve manner – dramatic speedups should be possible.

The Programming Language Shootout[2] has shown that low-level Haskell can compete 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 and presentation suggestions.

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*, pages 315–326. ACM Press, October 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.
8. Peter A. Jonsson and Johan Nordlander. Positive Supercompilation for a higher order call-by-value language. In *Proc. IFL 2007*, September 2007.
9. Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The essence of computation: complexity, analysis, transformation*, pages 379–403. Springer-Verlag, 2002.
10. Simon Marlow. *Deforestation for Higher-Order Functional Programs.* PhD thesis, University of Glasgow, 1996.

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

11. Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proc. ICFP '07*, pages 277–288. ACM Press, October 2007.
12. John Meacham. jhc: John's haskell compiler. `http://repetae.net/john/computer/jhc/`, 2007.
13. Neil Mitchell and Colin Runciman. Supero: Making Haskell faster. In *Proc. IFL 2007*, September 2007.
14. Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2007.
15. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *JFP*, 2(2):127–202, 1992.
16. 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.
17. Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
18. Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proc. ICFP '07*, pages 327–337. ACM Press, October 2007.
19. 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.
20. Simon Peyton Jones, Will Partain, and Andre Santos. Let-floating: Moving bindings to give faster programs. In *Proc. ICFP '96*, pages 1–12. ACM Press, 1996.
21. 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.
22. Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In *Proc. FPCA '95*, pages 282–292. ACM Press, 1995.
23. Jens Peter Secher and Morten Heine B. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *LNCS*, pages 113–127. Springer-Verlag, 2000.
24. M. Heine Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
25. The GHC Team. The GHC compiler, version 6.8. `http://www.haskell.org/ghc/`, November 2007.
26. Andrew Tolmach. An External Representation for the GHC Core Language. `http://www.haskell.org/ghc/docs/papers/core.ps.gz`, September 2001.
27. V F Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Copmutation*, pages 341–353. North-Holland, 1988.
28. V. F. Turchin. *Refal-5, Programming Guide & Reference Manual*. New England Publishing Co., Holyoke, MA, 1989.
29. Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.
30. Valentin F. Turchin, Robert M. Nirenberg, and Dimitri V. Turchin. Experiments with a supercompiler. In *Proc. LFP '82*, pages 47–55. ACM, 1982.
31. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc ESOP '88*, volume 300 of *LNCS*, pages 344–358. Berlin: Springer-Verlag, 1988.
32. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.