

# Rethinking Supercompilation

Neil Mitchell

ndmitchell@gmail.com

## Abstract

Supercompilation is a program optimisation technique that is particularly effective at eliminating unnecessary overheads. We have designed a new supercompiler, making many novel choices, including different termination criteria and handling of let bindings. The result is a supercompiler that focuses on simplicity, compiles programs quickly and optimises programs well. We have benchmarked our supercompiler, with some programs running more than twice as fast than when compiled with GHC.

**Categories and Subject Descriptors** D.3 [Software]: Programming Languages

**General Terms** Languages

**Keywords** Haskell, optimisation, supercompilation

## 1. Introduction

Consider a program that counts the number of words read from the standard input – in Haskell (Peyton Jones 2003) this can be compactly written as:

```
main = print ◦ length ◦ words ≪≪ getContents
```

Reading the program right to left, we first read the standard input as a string (`getContents`), then split it in to words (`words`), count the number of words (`length`), and print the result (`print`). An equivalent C program is unlikely to use such a high degree of abstraction, and is more likely to get characters and operate on them in a loop while updating some state.

Sadly, such a C program is three times faster, even using the advanced optimising compiler GHC (The GHC Team 2009). The abstractions that make the program concise have a significant runtime cost. In a previous paper (Mitchell and Runciman 2008) we showed how supercompilation can remove these abstractions, to the stage where the Haskell version is *faster* than the C version (by about 6%). In the Haskell program after optimisation, all the intermediate lists have been removed, and the `length ◦ words` part of the pipeline is translated into a state machine.

One informal description of supercompilation is that you simply “run the program at compile time”. This description leads to two questions – what happens if you are blocked on information only available at runtime, and how do you ensure termination? Answering these questions provides the design for a supercompiler.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP 2010

Copyright © ACM [to be supplied]...\$5.00

## 1.1 Contributions

Our primary contribution is the design of a new supercompiler (§2). Our supercompiler has many differences from previous supercompilers (§5.1), including a new core language, a substantially different treatment of let expressions and an entirely new termination criteria. The result is a supercompiler with a number of desirable properties:

**Simple** Our supercompiler is designed to be simple. From the descriptions given in this paper a reader should be able to write their own supercompiler. We have written a supercompiler following our design which is available online<sup>1</sup>. Much of the code (for example Figure 2) has been copied verbatim in to our implementation. The supercompiler can be implemented in under 300 lines of Haskell.

**Fast compilation** Previous supercompilers have reported compilation times of up to five minutes for small examples (Mitchell and Runciman 2008). Our compilation times are under four seconds, and there are many further compile time improvements that could be made (§4.2).

**Fast runtime** We have benchmarked our supercompiler on a range of small examples (§4). Some programs optimised with our supercompiler, and then compiled with GHC, are more than twice as fast than when compiled with GHC alone.

We give several examples of how our supercompiler performs (§2.3.2), including how it subsumes list fusion and specialisation (§3), and what happens when the termination criteria are needed (§2.6.4).

## 2. Method

This section describes our supercompiler. We first present a Core language (§2.1), along with simplification rules (§2.2). We then present the overall algorithm (§2.3), which combines the answers to the following questions:

- How do you evaluate an open term? (§2.4)
- What happens if you can’t evaluate an open term further? (§2.5)
- How do you know when to stop? (§2.6)
- What happens if you have to stop? (§2.6.3)

Throughout this section we use the following example:

```
root g f x = map g (map f x)
```

```
map f [] = []
map f (x : xs) = f x : map f xs
```

Our supercompiler always optimises the function named `root`. The `root` function applies `map` twice – the expression `map f x`

<sup>1</sup><http://hackage.haskell.org/package/supero>

```

type Var = String -- variable/function names
type Con = String -- constructor names

data Exp = App Var [Var] -- function application
        | Con Con [Var] -- constructor application
        | Let [(Var, Exp)] Var -- let expression
        | Case Var [(Pat, Exp)] -- case expression
        | Lam Var Exp -- lambda expression

type Pat = Exp -- restricted to Con

```

Figure 1. Core Language

produces a list that is immediately consumed by `map g`. A good supercompiler should remove the intermediate list.

## 2.1 Core Language

Our Core language for expressions is given in Figure 1, and has much in common with Administrative Normal Form (Flanagan et al. 1993). We make the following observations:

- We require variables in many places that would normally permit expressions, including let bodies and application. A standard Core language (such as from Tolmach (2001)) can be translated to ours by inserting let expressions.
- Our let expression is non-recursive – bindings within a let expression are bound in order. For example, we allow `let x = y; y = C in C` but not `let x = y; y = x in C`.
- We don't have default patterns in case expressions. These can be added without great complexity, but are of little interest when describing a supercompiler.
- We assume programs in our Core language are well-typed using Hindley-Milner, in particular that we never over-apply a constructor or perform case analysis on a function. While most Haskell programs can be translated to our Core language, the typing restriction means some features are not supported (GADTs, existential types).
- Function application takes a list of arguments, rather than just a single argument – the reasons are explained in §2.8.1. We use an application with no arguments to represent just a variable.
- Variables may be either local (bound in an expression), or global (bound in a top-level environment). We require that all global variables occur as the first argument of App.
- When comparing expressions we always normalise local variable names and the order of let bindings.

We define the arity of a variable to be the number of arguments that need to be applied before reduction takes place. For our purposes, it is important that for a variable with arity  $n$ , if less than  $n$  arguments are applied, no evaluation occurs. We approximate the arity of bound variables using the number of lambda arguments at the root of their expression, for primitives we use a known arity (e.g. integer addition has arity 2), and for all other variables we use arity 0. In our example `map` has arity 2, `root` has arity 3, and `f`, `g` and `x` have arity 0.

We write expressions using standard Haskell syntax (e.g. `let` for Let, `case` for Case etc.). Rewriting the `map/map` example in our Core language gives:

```

root = λg f x → let v1 = map f x
                v2 = map g v1
                in v2

```

```

map = λf x → case x of
            [] → let v1 = []
                in v1
            y : ys → let v1 = f y
                    v2 = map f y
                    v3 = (:) v1 v2
                in v3

```

Our Core language can be rather verbose, so we sometimes use a superset of our Core language, assuming the expressions are translated to our Core language when necessary. For example, we might write `map` as:

```

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

```

## 2.2 Simplified Core

We now define a simplified form of our Core language. When working with Core expressions we assume they are always simplified, and after constructing new expressions we always simplify them. We require all expressions bound in the top-level environment consist of a (possibly empty) sequence of lambda expressions, followed by a let expression. We call the first let expression within a top-level definition the *root let*.

Within a let we remove any bindings that are not used and ensure all bound variables are unique. We also require that all expressions bound at a let must *not* have the following form:

- `App v []`, where  $v$  is bound at this let – we can remove the binding by inlining it.
- `App v vs`, where  $v$  is bound to a Con – the App can be replaced with a Con of higher arity.
- `App v vs`, where  $v$  is bound to `App w ws` and the arity of  $w$  is higher than the length of  $ws$  – the App can be replaced with an App with more arguments.
- `App v vs`, where  $v$  is bound to a Lam – the App can be replaced with the body of the lambda, with the first variable substituted.
- `Case v w`, where  $v$  is bound to a Con – the Case can be replaced with the appropriate alternative.
- `Let bs v` – the bindings can be lifted into the let, renaming variables if necessary.

As an example, we can simplify the following expression:

```

let v1 = f
    v2 = Con x
    v3 = v2 y
    v4 = let w1 = y in v1 w1
    v5 = case v3 of Con a b → v4 a
in v5

```

To give:

```

let v4 = f y
    v5 = v4 x
in v5

```

If the arity of `f` was known to be 2, this would further simplify to:

```

let v5 = f y x
in v5

```

### 2.2.1 Simplifier Non-Termination

Sadly, not all expressions have a simplified form. Take the following example:

```

type Env = Var → Maybe Exp
data Tree = Tree
  { pre :: Exp, gen :: [Var] → Exp, children :: [Tree] }

manager :: Env → [(Var, Exp)]
manager env = assign (flatten (optimise env e))
  where Just e = env "root"

optimise :: Env → Exp → Tree
optimise env = f []
  where f h e | terminate (≤) h e = g h e (stop h e)
             | otherwise = g (e : h) e (reduce env e)
             g h e (gen, cs) = Tree e gen (map (f h) cs)

reduce :: Env → Exp → ([Var] → Exp, [Exp])
reduce env = f []
  where f h e = case step env e of
    _ | terminate (<) h e → stop h e
    Just e' → f (e : h) e'
    Nothing → split e

flatten :: Tree → [Tree]
flatten = nubBy (λt1 t2 → pre t1 ≡ pre t2) ∘ f []
  where f seen t = if pre t ∈ seen then [] else
    t : concatMap (f (pre t : seen)) (children t)

assign :: [Tree] → [(Var, Exp)]
assign ts = [(f t, gen t (map f (children t))) | t ← ts]
  where f t = fromJust (lookup (pre t) names)
        names = zip (map pre ts) functionNames

```

**Figure 2.** The manager function.

```

data U = MkU (U → Bool)
e = let f = λx → case x of MkU y → y × in f (MkU f)

```

This program encodes recursion via a data type, and any attempt to apply all the simplification rules will not terminate. We are aware of two solutions: 1) We could avoid performing case elimination on contravariant data types (Peyton Jones and Marlow 2002); 2) We could avoid simplifying certain expressions, using the size measure from the HOSC supercompiler (Klyuchnikov 2010).

We have chosen to leave this problem unsolved. The problem only occurs for contrived programs which encode recursion via a data type, and it is a problem shared with GHC, which will also non-terminate when compiling this example. The later stages of our supercompiler do not rely on the simplifications having been performed, so either solution could be applied in future.

### 2.3 Manager

Our supercompiler is based around a manager, that integrates the answers to the questions of supercompilation. The manager itself has two main purposes: to create recursive functions, and to ensure termination (assuming the simplifier terminates). In our experience the creation of recursive functions is often the most delicate part of a supercompiler, so we deliberately include all the details. The code for our manager is given in Figure 2, making use of a some auxiliary functions whose types are given in Figure 3. We first give an intuition for how the manager works, then describe each part.

Our supercompiler takes a source program, and generates a target program. Functions in these programs are distinct – target expressions cannot refer to source functions. The source and tar-

```

step :: Env → Exp → Maybe Exp -- §2.4
split :: Exp → ([Var] → Exp, [Exp]) -- §2.5

type History = [Exp]
(<), (≤) :: Exp → Exp → Bool -- §2.6
terminate :: (Exp → Exp → Bool)
  → History → Exp → Bool -- §2.6
stop :: History → Exp → ([Var] → Exp, [Exp]) -- §2.6.3

```

**Figure 3.** Auxiliary definitions for Figure 2.

get program are equivalent, but hopefully the target program runs faster. We use the type `Env` to represent a mapping from source function names to expressions, allowing a result of `Nothing` to indicate a primitive function.

The manager first builds a tree (the type `Tree`), where each node has a source expression (`pre`) and an equivalent target expression. The target expression may call target functions, but these functions do not yet have names. Therefore, we store target expressions as a generator that when given the function names produces the target expression (`gen`), and a list of trees for the functions it calls (`children`). We then flatten this tree, ensuring identical functions are only included once, and assign names to each node before generating the target program. If a target function is recursive then the initial tree will be infinite, but the flattened tree will always be finite due to the termination scheme defined in §2.6.

**manager:** This function puts all the parts together. Reading from right to left, we first generate a potentially infinite tree by optimising the expression bound to the function `root`, we then flatten the tree to a finite number of functions, and finally assign names to each of the result functions.

**optimise:** This function constructs the tree of result functions. While the tree may be infinite, we demand that any infinite path from the root must encounter the same `pre` value more than once. We require that for any infinite sequence of distinct expressions `h`, there must exist an `i` such that `terminate (≤) (take i h) (h !! i + 1)` returns `True` (where `(!!)` is the Haskell list indexing operator). If we are forced to terminate we call `stop`, which splits the expression into several subexpressions. We require that `stop h` only produces subexpressions which pass the termination test, so that when `f` is applied to the subexpressions they all call `reduce`. If the termination criteria do not force us to stop, then we call `reduce` to evaluate the expression.

**reduce:** This function optimises an expression by repeatedly evaluating it with calls to `step`. If we can't evaluate any further we call `split`. We use a local termination test to ensure the evaluation terminates. We require that for any infinite sequence of expressions `h`, there must exist an `i` such that `terminate (<) (take i h) (h !! i + 1)` returns `True`. Note that this termination criteria is more restrictive than that used by `optimise`.

**flatten:** This function takes a tree and extracts a finite number of functions from it, assuming the termination restrictions given in `optimise`. Our `flatten` function will only keep one tree associated with each source expression. These trees may have different target expressions if one resulted from a call to `stop`, while another resulted from a call to `reduce` – but all are semantically equivalent.

**assign:** This function assigns names to each target function, and constructs the target expressions by calling `gen`. We assume the function `functionNames` returns an infinite list of function names.

### 2.3.1 Notation

Values of type  $([Var] \rightarrow Exp, [Exp])$  can be described by taking an expression and removing some subexpressions (indicated by  $\llbracket \bullet \rrbracket$ ). The first component is a function to insert variables where subexpressions were removed, and the second component is a list of the removed subexpressions. Before removing each subexpression, we insert an applied lambda for all variables bound in the expression but free in the subexpression. As an example:

$\lambda g f \rightarrow \text{map } g (\llbracket \text{map } f \text{ xs} \rrbracket)$

We first insert a lambda for the variable  $f$ :

$\lambda g f \rightarrow \text{map } g (\llbracket (\lambda f \rightarrow \text{map } f \text{ xs}) f \rrbracket)$

We then remove the subexpression. The first component of the result is a function that when given  $["name"]$  returns the expression:

$\lambda g f \rightarrow \text{map } g (\text{name } f)$

And the second component is the singleton list containing the expression:

$\lambda f \rightarrow \text{map } f \text{ xs}$

### 2.3.2 Example

Revisiting our initial example, manager first calls `optimise` with:

$\lambda g f x \rightarrow \text{map } g (\text{map } f \text{ xs})$

The termination history is empty, so we call `reduce`, which calls `step` repeatedly until we reach the expression:

$\lambda g f x \rightarrow \text{let } v = \text{case } w \text{ of}$   
 $\quad [] \rightarrow []$   
 $\quad y : ys \rightarrow g y : \text{map } g \text{ ys}$   
 $w = \text{case } x \text{ of}$   
 $\quad [] \rightarrow []$   
 $\quad z : zs \rightarrow f z : \text{map } f \text{ zs}$   
**in**  $v$

The step function now returns `Nothing`, since we cannot evaluate further without the result of  $x$ . We therefore call `split`, which results in (using the notation from §2.3.1):

$\lambda g f x \rightarrow \text{case } x \text{ of}$   
 $\quad [] \rightarrow \llbracket \text{let } v = \dots; w = \dots; x = [] \text{ in } v \rrbracket$   
 $\quad z : zs \rightarrow \llbracket \text{let } v = \dots; w = \dots; x = z : zs \text{ in } v \rrbracket$

Looking at the first child expression, where  $x = []$ , the simplification rules from §2.2 immediately produce  $[]$  as the result. The second child starts as:

$\lambda g f z zs \rightarrow$   
 $\quad \text{let } x = z : zs$   
 $\quad w = \text{case } x \text{ of } [] \rightarrow []; z : zs \rightarrow f z : \text{map } f \text{ zs}$   
 $\quad v = \text{case } w \text{ of } [] \rightarrow []; y : ys \rightarrow g y : \text{map } g \text{ ys}$   
**in**  $v$

Which simplifies to:

$\lambda g f z zs \rightarrow \text{let } y = f z$   
 $\quad ys = \text{map } f \text{ zs}$   
 $\quad q = g y$   
 $\quad qs = \text{map } g \text{ ys}$   
 $\quad v = q : qs$   
**in**  $v$

Calling `step` produces `Nothing`, as the root of this expression is a constructor  $(:)$  which can't be evaluated. We therefore call `split` which results in:

$\text{force} :: Exp \rightarrow \text{Maybe } Var$   
 $\text{force } (\text{Case } v \_) = \text{Just } v$   
 $\text{force } (\text{App } v \_) = \text{Just } v$   
 $\text{force } \_ = \text{Nothing}$

$\text{next} :: Exp \rightarrow \text{Maybe } Var$   
 $\text{next } (\text{Lam } \_ x) = \text{next } x$   
 $\text{next } (\text{Let } \text{bind } v) = \text{last } (\text{Nothing} : f v)$   
 $\quad \text{where } f v = \text{case } \text{lookup } v \text{ bind of}$   
 $\quad \quad \text{Nothing} \rightarrow []$   
 $\quad \quad \text{Just } e \rightarrow \text{Just } v : \text{maybe } [] f (\text{force } e)$

**Figure 4.** Function to determine the next evaluated binding.

$\lambda g f z zs \rightarrow \text{let } q = \llbracket g (f z) \rrbracket$   
 $\quad qs = \llbracket \text{map } g (\text{map } f \text{ zs}) \rrbracket$   
 $\quad v = q : qs$   
**in**  $v$

When optimising  $g (f z)$  we get no optimisation, as there is no available information. To optimise  $\text{map } g (\text{map } f \text{ zs})$  we repeat the exact same steps we have already done. However, the `flatten` function will spot that both `Tree` nodes have the same pre expression (modulo variable renaming), and reduce them to one node, creating a recursive function. We then assign names using `assign`. For the purposes of display (not optimisation), we apply a number of simplifications given in §2.7. The end result is:

$\text{root } g f x = \text{case } x \text{ of}$   
 $\quad [] \rightarrow []$   
 $\quad z : zs \rightarrow g (f z) : \text{root } g f \text{ zs}$

The final version has automatically removed the intermediate list, with no additional knowledge about the `map` function or its fusion rules.

### 2.4 Evaluation

Evaluation is based around the `step` function. Given an expression, `step` either replaces a variable with its associated value from the environment and returns `Just`, or if no suitable variable is found returns `Nothing`. We always replace the variable that would be evaluated next during normal evaluation.

To determine which variable would be evaluated next, we define the functions `force` and `next` in Figure 4. The function `force` determines which variable will be evaluated next given an expression – either a case scrutinee or an applied variable. The function `next` determines which variable bound at the root `let` will be evaluated next, by following the forced variables of the `let` bindings. Looking at the original example:

$\lambda g f x \rightarrow \text{let } v_1 = \text{map } f \text{ xs}$   
 $\quad v_2 = \text{map } g \text{ v}_1$   
**in**  $v_2$

The function `next` returns `Just v2`. Calling `force` on the expression  $\text{map } g \text{ v}_1$  returns `map`, but since `map` is not bound at the root `let` we go no further. Therefore, to evaluate this expression we will start by evaluating  $v_2$ , and thus `map`. To perform an evaluation step we insert a fresh variable  $v_1$  bound to the body of `map`, and replace the `map` variable in  $v_2$  with  $v_1$ . This transformation results in:

$\lambda g f x \rightarrow \text{let } v_1 = \text{map } f \text{ xs}$   
 $\quad w_1 = \lambda f x \rightarrow \text{case } x \text{ of}$   
 $\quad \quad [] \rightarrow []$   
 $\quad \quad y : ys \rightarrow f y : \text{map } f \text{ ys}$

$$\begin{array}{l} v_2 = w_1 g v_1 \\ \text{in } v_2 \end{array}$$

Simplification immediately removes the lambda at  $w_1$ , replacing  $v_2$  with a case expression on  $v_1$ .

More generally, we match any expression with the following pattern:

$$\begin{array}{l} \lambda \text{free} \rightarrow \text{let } s = f w_1 w_n \\ \quad v_1 = e_1 \\ \quad \quad v_n = e_n \\ \quad \text{in } v \\ \text{where Just } e' = \text{env } f \end{array}$$

We use  $s$  to represent the next binding to be evaluated, as returned by next. We allow any other variables  $v_1 \dots v_n$  to be present, bound to expressions  $e_1 \dots e_n$ . Given this configuration we can rewrite to:

$$\begin{array}{l} \lambda \text{free} \rightarrow \text{let } s' = e' \\ \quad s = s' w_1 w_n \\ \quad v_1 = e_1 \\ \quad \quad v_n = e_n \\ \quad \text{in } v \end{array}$$

As always, after generating a new expression we immediately apply the simplification rules (§2.2).

## 2.5 Evaluation Splitting

If evaluation cannot proceed, we split to produce a target expression, and a list of child expressions for further optimisation. When splitting an expression there are three concerns:

**Permit further optimisation:** When we split, the current expression cannot be evaluated using the rules described in §2.4. We therefore aim to place the construct blocking evaluation in the target expression, allowing the child expressions to be optimised further.

**No unbounded loss of sharing:** An expensive variable binding cannot be duplicated in a way that causes it to be evaluated multiple times at runtime. The target program cannot remove sharing present in the source program, or it would run slower.

**Keep bindings together:** If we split variables bound at the same let expression into separate child expressions, we reduce the potential for optimisation. If the expression associated with a variable is not available when evaluating, the evaluation will be forced to stop sooner. We aim to make child expressions as large as possible, but without losing sharing.

We split in one of three different ways, depending on the type of the next expression to be evaluated (as described in §2.4). We now describe each of the three ways to split, in each case we start with an example, then define the general rule. We use the  $\llbracket \bullet \rrbracket$  notation described in §2.3.1.

### 2.5.1 Case Expression

If the next expression is a case expression then we make the target a similar case expression, and under each alternative we create a child expression with the case scrutinee bound to the appropriate pattern. For example, given:

$$\begin{array}{l} \lambda x \rightarrow \text{let } v = \text{case } x \text{ of} \\ \quad \quad \quad \llbracket \quad \rrbracket \rightarrow \llbracket \quad \rrbracket \\ \quad \quad \quad y : ys \rightarrow \text{add } y \text{ } ys \\ \quad \text{in } v \end{array}$$

We split to produce:

$$\lambda x \rightarrow \text{case } x \text{ of} \\ \quad \llbracket \quad \rrbracket \rightarrow \llbracket \text{let } x = \llbracket \quad \rrbracket$$

$$\begin{array}{l} v = \text{case } x \text{ of } \llbracket \quad \rrbracket \rightarrow \llbracket \quad \rrbracket; y : ys \rightarrow \text{add } y \text{ } ys \\ \text{in } v \\ y : ys \rightarrow \llbracket \text{let } x = y : ys \\ \quad v = \text{case } x \text{ of } \llbracket \quad \rrbracket \rightarrow \llbracket \quad \rrbracket; y : ys \rightarrow \text{add } y \text{ } ys \\ \quad \text{in } v \rrbracket \end{array}$$

Looking more closely at the second child, we start with the expression:

$$\begin{array}{l} \lambda y \text{ } ys \rightarrow \text{let } x = y : ys \\ \quad v = \text{case } x \text{ of } \llbracket \quad \rrbracket \rightarrow \llbracket \quad \rrbracket; y : ys \rightarrow \text{add } y \text{ } ys \\ \quad \text{in } v \end{array}$$

This expression immediately simplifies to:

$$\lambda y \text{ } ys \rightarrow \text{let } v = \text{add } y \text{ } ys \\ \quad \text{in } v$$

More generally, if  $s$  is the next expression to evaluate:

$$\begin{array}{l} \lambda \text{free} \rightarrow \text{let } s = \text{case } x \text{ of } p_1 \rightarrow e'_1; p_m \rightarrow e'_m \\ \quad v_1 = e_1 \\ \quad \quad v_n = e_n \\ \quad \text{in } v \end{array}$$

After split it becomes:

$$\begin{array}{l} \lambda \text{free} \rightarrow \text{case } x \text{ of} \\ \quad p_1 \rightarrow \llbracket \text{let } x = p_1 \\ \quad \quad s = e'_1 \\ \quad \quad v_1 = e_1 \\ \quad \quad \quad v_n = e_n \\ \quad \quad \text{in } v \rrbracket \\ \quad p_m \rightarrow \llbracket \text{let } x = p_m \\ \quad \quad s = e'_m \\ \quad \quad v_1 = e_1 \\ \quad \quad \quad v_n = e_n \\ \quad \quad \text{in } v \rrbracket \end{array}$$

### 2.5.2 Lambda

If the next binding to be evaluated is a lambda, then we place a lambda in the target program. The key point when splitting a lambda is that we do not reduce sharing. Consider the following example:

$$\begin{array}{l} \lambda x \rightarrow \text{let } v_1 = f x \\ \quad v_2 = \text{expensive } v_1 \\ \quad s = \lambda y \rightarrow \text{add } v_2 \text{ } y \\ \quad \text{in } s \end{array}$$

The add function takes two arguments, but only has one so far. We cannot move the argument  $y$  upwards to form  $\lambda x \text{ } y \rightarrow \dots$ , as this action potentially duplicates the expensive computation of  $v_2$ . Instead, we create child expressions for every variable binding, and for the body of the lambda:

$$\begin{array}{l} \lambda x \rightarrow \text{let } v_1 = \llbracket f x \rrbracket \\ \quad v_2 = \llbracket \text{expensive } v_1 \rrbracket \\ \quad s = \lambda y \rightarrow \llbracket \text{add } v_2 \text{ } y \rrbracket \\ \quad \text{in } s \end{array}$$

Unfortunately, we have now split the bindings for  $v_1$  and  $v_2$  apart, when there is no real need. We therefore move binding  $v_1$  under  $v_2$ , because it is only referred to by  $v_2$ , to give:

$$\begin{array}{l} \lambda x \rightarrow \text{let } v_2 = \llbracket \text{let } v_1 = f x \text{ in expensive } v_1 \rrbracket \\ \quad s = \lambda y \rightarrow \llbracket \text{add } v_2 \text{ } y \rrbracket \\ \quad \text{in } s \end{array}$$

We will now optimise the body of  $v_2$ , and the body of the lambda, which will be able to evaluate add. More generally, given:

$$\lambda \text{free} \rightarrow \text{let } s = \lambda x \rightarrow e' \\ \quad v_1 = e_1 \\ \quad v_n = e_n \\ \text{in } v$$

We rewrite:

$$\lambda \text{free} \rightarrow \text{let } s = \lambda x \rightarrow \llbracket e' \rrbracket \\ \quad v_1 = \llbracket e_1 \rrbracket \\ \quad v_n = \llbracket e_n \rrbracket \\ \text{in } v$$

We then repeatedly move any binding  $v_i$  under  $v_j$  if either: 1)  $v_i$  is only used within the body of  $v_j$ ; or 2) the expression bound to  $v_i$  is cheap. We define an expression to be cheap if it is a constructor, or an application to a variable  $v$  with fewer arguments than the arity of  $v$  (a partial application). The intention of moving bindings is to increase sharing, which can be done provided we don't duplicate work (condition 1), or if the work duplicated is bounded (condition 2).

### 2.5.3 Anything Else

The final rule applies if the next expression is not a case expression or a lambda, including a constructor, a variable, and an application of a variable not bound in the environment. We do not deal with variables bound in the environment, as these are handled by step. Given the example:

$$\lambda x y \rightarrow \text{let } v_1 = \text{expensive } x \\ \quad v_2 = v_1 \times \\ \quad v_3 = \text{Con } v_2 y v_2 \\ \text{in } v_3$$

We turn each binding into a child, apart from the next binding to be evaluated:

$$\lambda x y \rightarrow \text{let } v_1 = \llbracket \text{expensive } x \rrbracket \\ \quad v_2 = \llbracket v_1 \times \rrbracket \\ \quad v_3 = \text{Con } v_2 y v_2 \\ \text{in } v_3$$

We then perform the same sharing transformation as for lambda expressions, noting that  $v_1$  is only used within  $v_2$ , to give:

$$\lambda x y \rightarrow \text{let } v_2 = \llbracket \text{let } v_1 = \text{expensive } x \text{ in } v_1 \times \rrbracket \\ \quad v_3 = \text{Con } v_2 \times v_2 \\ \text{in } v_3$$

More generally, given an expression:

$$\lambda \text{free} \rightarrow \text{let } s = e' \\ \quad v_1 = e_1 \\ \quad v_n = e_n \\ \text{in } v$$

We rewrite to:

$$\lambda \text{free} \rightarrow \text{let } s = e' \\ \quad v_1 = \llbracket e_1 \rrbracket \\ \quad v_n = \llbracket e_n \rrbracket \\ \text{in } v$$

We then repeatedly move any binding  $v_i$  under  $v_j$  according to the criteria given in §2.5.2.

## 2.6 Termination

The termination rule is responsible for ensuring that whenever we proceed along a list of expressions we eventually stop. The intuition

is that each expression has a set of bindings at the root let, and each of these bindings has a name indicating where it came from in the source program. Compared to all earlier expressions in a list, each root let must contain either different names, or fewer names.

In this section we first describe the terminate,  $\triangleleft$  and  $\trianglelefteq$  functions from a mathematical perspective, then how we apply these functions to expressions. Finally, we show an example of how these rules ensure termination.

### 2.6.1 Termination Rule

Our termination rules are defined over bags (also known as multisets) of values drawn from a finite alphabet  $\Sigma$ . A bag of values is unordered, but may contain elements more than once. We define our rules as:

$$x \triangleleft y = \text{set}(x) \neq \text{set}(y) \vee \#x < \#y \\ x \trianglelefteq y = x \equiv y \vee x \triangleleft y$$

We use  $\text{set}(x)$  to transform a bag to a set, and  $\#$  as the cardinality operator to take the number of elements in a bag. A sequence  $x_1 \dots x_n$  is well-formed under  $\triangleleft$  if for all indices  $i < j \Rightarrow x_j \triangleleft x_i$  (and respectively for  $\trianglelefteq$ ).

The following sequences are well-formed under both  $\trianglelefteq$  and  $\triangleleft$ :

```
[a, aaaaab, aaabb, b]
[abc, ab, accc, a]
[aaaaabbb, aaab, aab]
```

The following sequences are well-formed under  $\trianglelefteq$ , but not under  $\triangleleft$ :

```
[aaa, aaa]
[aabb, ab, ab]
```

The following sequences are not well-formed under  $\trianglelefteq$  or  $\triangleleft$ :

```
[abc, abcc]
[aa, aaa]
```

We define the terminate function referred to in Figure 3 as:

```
terminate :: (Exp → Exp → Bool) → History → Exp → Bool
terminate (<) h e = not (all (e<) h)
```

The terminate function returns False if given a well-formed sequence (h), adding the expression e will keep the sequence well-formed.

#### *Lemma: Any well-formed sequence under $\triangleleft$ is finite*

Given a finite alphabet  $\Sigma$ , any well-formed sequence under  $\triangleleft$  is finite. Consider a well-formed sequence  $x_1 \dots$ . We can partition this sequence into at most  $2^{\Sigma}$  subsequences using set equality. Consider any subsequence  $y_1 \dots$ . For any two elements in this subsequence,  $\text{set}(y_i) \neq \text{set}(y_j)$  will be false, due to the partitioning. Therefore, for the sequence to be well-formed,  $i < j \Rightarrow \#y_j < \#y_i$ . Therefore there can be at most  $\#y_1 + 1$  elements in any particular subsequence. Combined with a finite number of subsequences, we conclude that any well-formed sequence is finite.

#### *Lemma: Any well-formed sequence under $\trianglelefteq$ has a finite number of distinct elements.*

Given a finite alphabet  $\Sigma$ , any well-formed sequence under  $\trianglelefteq$  has a finite number of distinct elements. For a sequence to be well-formed under  $\trianglelefteq$  but not  $\triangleleft$  it must have elements which are duplicates. If we remove all duplicates we end up with a well-formed sequenced under  $\triangleleft$ , which must be finite. Therefore there must be a finite number of distinct elements.

### 2.6.2 Tracking Names

Every expression in the source program is assigned a name. A name is a triple,  $\langle f, e, a \rangle$  where  $f$  is a function name,  $e$  is an expression

index and  $a$  is an argument count. We label every expression in the source program with  $f$  being the function it comes from and  $e$  being a unique index within that function. The argument count  $a$  for constructors and applications is the number of arguments, and for all other expressions is 0. When manipulating expressions, we track names:

- When renaming a bound variable, or substituting one variable for another, we do not change any names.
- If we move a subexpression, we keep the name already assigned to that subexpression.
- If we increase the number of arguments to an application or constructor, we increase the argument count of that expression. For example,  $\mathbf{let\ } v = C\ x; w = v\ y\ \mathbf{in\ } w$  being transformed to  $\mathbf{let\ } w = C\ x\ y\ \mathbf{in\ } w$  would have the new name for  $w$  set to the old name of  $v$ , but with an argument count of 2 instead of 1.
- When splitting on a case we introduce a new constructor (see §2.5.1), for this constructor we use the name assigned to the pattern from the case alternative.

We map an expression to a bag of names by taking the names of all expressions bound at the root let.

**Lemma:** *For any source program, there are a finite number of names*

All subexpressions are assigned expression indices in advance, so there are only a finite number of function name/index values. We only increase the argument count when increasing the number of arguments applied to a constructor or application, which is bounded by the arity of that constructor or the source function. Therefore, there are only a finite number of names.

**Lemma:** *A bag of names represents a finite number of expressions*

Given a bag of names, there are only a finite number of expressions that could have generated it. We first assume that when simplifying an expression we always normalise the free variables – naming the let body  $v_1$ , and naming all other variables as they are reached from  $v_1$ . Each name refers to one particular subexpression, but may have different variable names. A finite number of subexpressions can only be combined to produce a finite number of expressions, if we ignore variable names, which the normalisation handles.

**Lemma:** *The termination properties required by §2.3 are satisfied*

The termination properties in §2.3 are satisfied by the lemmas in this section. We have shown that the alphabet of names,  $\Sigma$ , is finite. For terminate ( $\trianglelefteq$ ) we have shown that there can only be a finite number of distinct name bags, and that each name bag can only correspond to a finite number of expressions, therefore there are a finite number of distinct expressions. For terminate ( $\triangleleft$ ) we have shown that there can only be a finite number of name bags.

### 2.6.3 Termination Splitting

If we are forced to terminate we call stop, which splits the expression into several subexpressions. We require that stop  $h$  e only produces subexpressions which are not forced to terminate by terminate ( $\trianglelefteq$ )  $h$ . We trivially satisfy this requirement by using the termination criteria when defining stop.

Given an expression:

$$\lambda \text{free} \rightarrow \mathbf{let\ } v_1 = e_1 \\ \quad \quad \quad v_n = e_n \\ \mathbf{in\ } v$$

We first split every variable bound at the let, to give:

$$\lambda \text{free} \rightarrow \mathbf{let\ } v_1 = \llbracket e_1 \rrbracket \\ \quad \quad \quad v_n = \llbracket e_n \rrbracket \\ \mathbf{in\ } v$$

We now move variable  $v_i$  under  $v_j$  using the same conditions as split, described in §2.5.2. In addition, we do not merge  $v_i$  under  $v_j$  if the resulting expression bound to  $v_j$  would violate the termination criteria terminate ( $\trianglelefteq$ )  $h$ . Combined with the property that all initial children are singleton name bags, which trivially satisfy  $\trianglelefteq$  for any expression, our merge restriction ensures no children violate the termination criteria.

As a heuristic, we attempt to move variable  $v$  before  $w$  if the name associated with  $v$  occurs fewer times in the original expression. In most expressions that are growing, and therefore hit the termination criteria, there will be some name that keeps repeating. By favouring names that are less frequent we hope to keep together subexpressions that are not growing. This heuristic has no effect on the correctness or termination, but can sometimes result in better optimisation.

### 2.6.4 Example

Many simple example programs (such as map/map) never trigger the termination criteria. The standard example of a function that does require termination is reverse, which can be written in a simplified form as:

```
root xs = rev [] xs
rev acc xs = case xs of
    []      → acc
  y : ys   → rev (y : acc) ys
```

The rev function builds up an accumulator argument, which will be equal to the size of xs. To specialise on the accumulator argument would require an infinite number of specialisations. To supercompile this program, the optimise function starts with an empty termination history and the expression rev [] xs, and calls reduce, resulting in:

```
λxs → case xs of
    []      → [[]]
  y : ys   → [rev (y : []) ys]
```

Focusing on the second alternative, we now add rev [] xs to the termination history, and continue optimising rev (y : []) ys. This leads to the sequence of expressions:

```
λx1 → rev [] x1
λx1 x2 → rev (x1 : []) x2
λx1 x2 x3 → rev (x1 : x2 : []) x3
...
```

We can rewrite these expressions in our core language, with annotations for names:

```
λx1 →
  let v1 = ⟨root, 2, 0⟩ []
    v2 = ⟨root, 1, 0⟩ rev v1 x1
  in v2
λx1 x2 →
  let v1 = ⟨root, 2, 0⟩ []
    v2 = ⟨rev , 2, 0⟩ x1 : v1
    v3 = ⟨rev , 1, 0⟩ rev v2 x2
  in v1
λx1 x2 x3 →
  let v1 = ⟨root, 2, 0⟩ []
    v2 = ⟨rev , 2, 0⟩ x2 : v1
    v3 = ⟨rev , 2, 0⟩ x1 : v2
```

```

v4 = ⟨rev , 1, 0⟩ rev v3 x3
in v1

```

Under  $\sqsubseteq$  the first two expressions create a well-formed sequence, but the first three expressions do not. The first expression is permitted because the history is empty. The second expression is permitted because it has a different set of names from the first. The third expression has the same set of names as the second, and has a higher cardinality. Therefore, when optimising, we call stop on the third expression. After calling stop we get:

```

λx1 x2 x3 →
let v2 = [[let v1 = ⟨root, 2, 0⟩ []
          v2 = ⟨rev , 2, 0⟩ x2 : v1
          in v2]]
v4 = [[let v3 = ⟨rev , 2, 0⟩ x1 : v2
       v4 = ⟨rev , 1, 0⟩ rev v3 x3
       in v4]]
in v1

```

Part of the accumulator has been bound to  $v_2$ , and separated from the main expression. Continuing to optimise we get the sequence:

```

λx1 → rev [] x1           -- reduce
λx1 x2 → rev (x1 : []) x2 -- reduce
λx1 x2 x3 → rev (x1 : x2 : []) x3 -- stop
λx1 x2 x3 → rev (x1 : x2) x3 -- reduce
λx1 x2 x3 x4 → rev (x1 : x2 : x3) x4 -- stop
λx1 x2 x3 → rev (x1 : x2) x3 -- reduce
... -- repeat the last 2 lines

```

As required, we have a finite number of distinct expressions, and end up with a recursive function in the target program.

## 2.7 Post-processing

Our split function is structured to produce only one simple expression per target function – for example a target function will never contain two constructors. While most opportunities to remove intermediate structure have been exploited, the target program will usually contain lots of small functions. We can eliminate many of these functions by inlining all functions which are only called once. For example, given the source program:

```
root x = x : x : []
```

After supercompilation, we get the target program:

```

root x = x : f x
f x = x : nil
nil = []

```

We can then inline all functions that are only called once:

```
root x = x : x : []
```

It is important that the only optimisation intended from this post-processing is the reduction of function call overhead. This use of inlining is substantially different from other compilers (Peyton Jones and Marlow 2002), where inlining is used to bring expressions together to trigger other optimisations.

## 2.8 Alternative Designs

In this section we describe some possible design alternatives for our supercompiler.

### 2.8.1 Binary Application

The first version of this supercompiler had binary application, rather than vector application. The `App Var [Var]` constructor was replaced by a combination of `Var Var` and `App Var Var`. The

reason for originally choosing binary application is that it is closer to other Core languages, and the simplification does not need to track arity information. There were three main reasons for moving to vector application:

- Vector application simplifies splitting with primitive functions, by providing the arity information directly.
- Vector application makes it easier to identify partial applications when increasing sharing (see §2.5.2).
- Vector application reduces the number of names in an expression, improving the time taken to compile.

### 2.8.2 Alternative Termination Orderings

Our original termination rule was:

$$x \triangleleft y = x \supset_{\text{set}} y \vee x \subset_{\text{bag}} y$$

Both this rule and the one described in §2.6.1 can be proved using the same argument. We switched to use our new rule because it is simpler, follows more directly from the proof, and can be implemented very efficiently. Choosing a termination rule is a tricky business – no termination rule can be the best for all programs, so there is always scope for experimentation.

### 2.8.3 Recursive Lets

Our Core language does not include recursive lets. Recursive lets bound to functions can be efficiently removed using lambda-lifting (Johnsson 1985). Recursive lets bound to values can be removed, but doing so may cause the program to run arbitrarily slower (Mitchell 2008). Alternatively, we can take functions with value recursive lets and make them primitives, losing optimisation potential, but preserving complexity. In practice, the most common function with a value recursive let is `repeat`, and our supercompiler is nearly always able to fuse away the list generated by `repeat`.

### 2.8.4 Common Subexpression Elimination

Common Subexpression Elimination (CSE) involves detecting when a program will compute two identical expressions, and reducing them both to a single shared expression. Our Core language is well suited to CSE – two variables can be merged if they have the same bound expression. The advantage of CSE is that performance can be increased, sometimes asymptotically. The disadvantages are that CSE can introduce space leaks (Chitil 1998), and the additional sharing may stop variables from being moved when splitting. We have not investigated the performance impact of CSE on supercompilation, but think it is a worthwhile area for future research.

### 2.8.5 Inlining Simple Functions

The GHC compiler inlines many non-recursive functions during the simplification phase (Peyton Jones and Marlow 2002). It is certainly possible that our simplification rules could be extended to inline some functions, such as `id`, provided no new names were introduced (and thus termination was unaffected). Another alternative would be to inline simple functions in the source program before supercompilation started (such as `otherwise` and `(◦)`). The primary motivation for inlining simple functions would be to reduce the complexity of the main supercompilation phase, and avoid inopportune termination splits. We have deliberately *not* performed any inlining other than in the step function, as there is a risk that doing so would hide weaknesses in our supercompiler. However, we think simple inlining would be worth investigating.

## 3. Comparison to Other Optimisations

Supercompilation naturally subsumes many other optimisations, including constructor specialisation (Peyton Jones 2007) and deforestation (Gill et al. 1993; Wadler 1990). However, there are some

optimisations that supercompilation (in the form presented here) does not address – in particular strictness analysis and unboxing (Peyton Jones and Launchbury 1991), and the generation of native code. In order to benefit from these optimisations we use GHC to compile the resulting Core after supercompilation (The GHC Team 2009).

We now give an example where our supercompiler massively outperforms GHC, and discuss the optimisations being performed. Our example is:

```
root n = map square (iterate (+1) 1) !! n
  where square x = x * x
```

Running this program with  $n = 400000$ , GHC takes 0.149 seconds, while our supercompiler combined with GHC takes 0.011 seconds. Running for larger values of  $n$  is infeasible as the GHC only variant overflows the stack. After optimising with our supercompiler, then compiling with GHC, the resulting inner-loop is:

```
go :: Int# -> Int# -> Int #
go x y = case x of
  0 -> y * y
  _ -> go (x - 1) (y + 1)
```

All the intermediate lists have been removed, there are no functional values, all the numbers have been unboxed and all arithmetic is performed on unboxed values (GHC uses `Int#` as the type of unboxed integers). Supercompilation has fused all the intermediate lists and specialised all functional arguments, leaving GHC to perform strictness analysis and unboxing.

The program compiled with GHC alone is much less efficient. GHC uses programmer supplied rewrite rules to eliminate intermediate lists (Peyton Jones et al. 2001), which fuses the `map/iterate` combination. Unfortunately, GHC does not contain a rule to fuse the input list to the `(!!)` operator. The GHC rules match specific function names in the source program, meaning that redefining `map` locally would inhibit the fusion. In contrast, our supercompiler does not rely on rules so is able to fuse the functions regardless of their names, and is able to perform fusion on data types other than lists.

GHC specialises the resulting `map/iterate` combination with the `square` function, but fails to specialise with `increment` – passing `(+1)` as a higher-order function. GHC can specialise functions to particular data values using constructor specialisation, but does not currently do the same transformation for functional arguments. To allow specialisation, some functions are written in a particular style:

```
foldr f z xs = go xs
  where go [] = z
        go (y : ys) = f y (go ys)
```

In this definition, provided lambda-lifting is not performed, the function `foldr` is considered non-recursive. GHC can inline non-recursive functions, allowing the definition of `foldr` to be replicated in an expression where `f` is known, eliminating the functional argument. In contrast, our supercompiler has specialised all the functions to their functional arguments, even when written in a natural style.

GHC fails to eliminate all the lists and higher-order functions, which in turn means the integers are not detected as strict, and are not unboxed. In contrast, our supercompiler has reduced the program sufficiently for everything to be unboxed.

## 4. Benchmarks

In this section we run our supercompiler over a range of benchmarks drawn from other papers. The results are given in Table 1. The benchmarks we use are:

Program	Lines	Compile time	Runtime	Memory	Size
append	8	0.1 + 0.6	0.85	0.84	1.00
bernouilli	148	2.4 + 1.3	1.04	0.96	1.02
charcount	32	0.1 + 0.6	0.14	0.01	0.99
digits-of-e2	100	2.0 + 0.8	0.40	0.45	0.99
exp3_8	39	0.5 + 0.8	0.93	1.00	1.08
factorial	12	0.1 + 0.6	0.98	1.00	1.00
linecount	43	0.2 + 0.6	0.01	0.01	0.98
primes	58	0.1 + 0.6	0.58	0.81	0.99
raytracer	26	0.1 + 0.6	0.56	0.44	1.00
rfib	16	0.1 + 0.7	0.77	1.01	0.98
sumsquare	45	1.2 + 0.9	0.38	0.23	1.03
sumtree	27	0.1 + 0.6	0.14	0.01	1.00
tak	19	0.1 + 0.6	0.79	1.01	0.98
treeflip	26	0.1 + 0.6	0.57	0.45	1.01
wordcount	62	0.3 + 0.7	0.19	0.30	1.00
x2n1	36	0.1 + 0.8	0.90	0.99	1.00

**Program** is the name of the program as given in §4. **Lines** is the number of lines of code in the original program, including library definitions, but excluding primitives. **Compile time** is the number of seconds to compile the program ( $a + b$ ), including both compilation with our supercompiler ( $a$ ) and the subsequent compilation with GHC ( $b$ ). The final three columns are relative to `ghc -O2` being 1.00, with a lower number being better. **Runtime** is how long the optimised program takes to run. **Memory** is the amount of memory allocated on the heap. **Size** is the size of the optimised program on disk.

Table 1. Benchmark results.

- `sumsquare` is the introductory example used in the stream fusion paper (Coutts et al. 2007a).
- `charcount`, `linecount` and `wordcount` are taken from our previous supercompiler work (Mitchell and Runciman 2008), and `wordcount` is used as the example in §1. For the purpose of benchmarking, we have removed the actual IO operations, leaving just the actual computation.
- `append`, `factorial`, `treeflip`, `sumtree` and `raytracer` have been used to benchmark other supercompilers (Jonsson and Nordlander 2009), and originate from papers on deforestation (Wadler 1990; Kort 1996).
- `bernouilli`, `digits-of-e2`, `exp3_8`, `primes`, `rfib`, `tak` and `x2n1` are all taken from the Imaginary section of the `nofib` benchmark suite (Partain et al. 2008).

We have manually translated all the examples from their source language to our Core language. We have taken care to ensure that we have not simplified the programs in translation – in particular we have inserted explicit dictionaries for all examples that require type classes (Wadler and Blott 1989), and have translated list-comprehensions to `concatMap` as described by the Haskell report (Peyton Jones 2003).

For comparison purposes we compiled all the benchmarks with GHC 6.12.1 (The GHC Team 2009), using the `-O2` optimisation setting. For the supercompiled results we first ran our supercompiler, then compiled the result using GHC. To run the benchmarks we used a 32bit Windows machine with a 2.5GHz processor and 4Gb of RAM.

### 4.1 Comparison to GHC

The benchmarks are nearly all faster than GHC, with some programs running substantially faster than GHC alone. The improvement in speed is usually accompanied by either a similar memory usage, or a substantial reduction. The resulting executables are all

very close in size to compilation with GHC alone – partly because the run-time system accounts for a substantial proportion of the executable size.

**Numerical Computation** Some of the benchmarks mainly test numerical performance – for example factorial, x2n1 and tak. In these benchmarks we have been able to inline some of the functions even though they are recursive, which has been equivalent to a small amount of loop unrolling, and has sometimes improved execution speed.

**Complete Elimination** Some of the benchmarks allow us to completely eliminate most intermediate values – for example charcount, sumtree and raytracer. In these cases the execution time and memory are both substantially reduced. Most of these benchmarks have previously been used to test supercompilers, and our supercompiler performs the same optimisations.

**Partial Elimination** Some of the benchmarks have a combination of data structures and numerical computation – for example primes, digits-of-e2 and exp3.8. In these benchmarks we perform specialisation, and remove some intermediate values, but due to the nature of the benchmarks not all intermediate values can be eliminated. In digits-of-e2 we are able to fuse long pipelines of list fusions. In exp3.8 most of our performance increase comes from eliminating intermediate values of the data type `data Nat = Z | S Nat`.

**Bernoulli** The benchmark on which we do worst is bernouilli. The bernouilli program seems reasonably similar in terms of list operations to other benchmarks such as primes, but our supercompiler is unable to outperform GHC – the exact reasons are still unclear. Interestingly, both our previous supercompiler and the stream fusion work also failed to outperform GHC on this benchmark, so the reason may be that GHC does a particularly good job on this benchmark.

## 4.2 Compilation Speed

In the benchmarks presented, our supercompiler always takes under four seconds to compile. We have given the compilation times as two figures – the time taken to run the supercompiler, followed by the time taken to compile the result with GHC. In all cases, the resulting GHC compilation time is dominated by the linker. Compared to our previous supercompiler, where compile times ranged from a few seconds to five minutes, our new supercompiler is substantially faster.

While we have designed our supercompiler with compilation speed in mind, we haven't focused on optimising the compiler – all functions are implemented as simply as possible. Profiling shows that 80% of the compilation time is spent simplifying expressions, as described in §2.2. Our simplification method is currently written as a transformation that is applied until a fixed point is reached – we believe the simplification can be implemented in one pass, leading to a substantial reduction in compile time.

We have implemented the termination check exactly as described in §2.6.1, traversing and comparing the entire history at each step. For our termination check it is simple to change the history to a mapping from a name bag to an integer (being the highest permitted cardinality) – reducing the algorithmic complexity. We could also optimise the representation of names, using a single integer for both the function name and subexpression index.

## 5. Related Work

We first describe related work in the area of supercompilation, particularly what makes our supercompiler unique. We then describe some work from other areas, particularly work from which we have used ideas.

### 5.1 Supercompilation

Supercompilation was introduced by Turchin (1986) for the Refal programming language (Turchin 1989). Since this original work, there have been many suggestions of both termination strategies and generalisation/splitting strategies (Turchin 1988; Sørensen and Glück 1995; Leuschel 2002). The original supercompiler maintained both positive and negative knowledge (Secher and Sørensen 2000), however our implementation uses only positive knowledge (what a variable is, rather than what it cannot be). More recently, supercompilation has started to converge towards a common design, described in detail by Klyuchnikov (2009), but which has much in common with the underlying design present in other papers (Mitchell and Runciman 2008; Jonsson and Nordlander 2009).

Compared to an increasingly common foundation, our supercompiler is radically different. We have changed many of the ingredients of supercompilation (the treatment of let, termination criteria, how the termination histories are used), but have also changed the way these ingredients are combined (the manager). In particular, many of our choices would not work if applied in isolation to another supercompiler – for example the termination criteria relies on the treatment of let.

#### 5.1.1 Let Expressions

Compared to other supercompilers, our Core language requires many more let expressions. Previous supercompilation work has tended to ignore let expressions – if let is mentioned the usual strategy is to substitute all linear lets and residue all others. At the same time, movement of lets can have a dramatic impact on performance: carefully designed let-shifting transformations give an average speedup of 15% in GHC (Peyton Jones et al. 1996).

Our previous work inlined all let bindings that it could show did not lead to a loss of sharing (Mitchell and Runciman 2008). Unfortunately, where a let could not be removed, there was a substantial performance penalty. By going to the opposite extreme we are forced to deal with let bindings properly, making our new supercompiler both simpler and more robust.

#### 5.1.2 Termination Criteria

The standard termination criteria used by supercompilers is the homeomorphic embedding (Leuschel 2002). The homeomorphic embedding is a well-quasi ordering, from Kruskal's Tree Theorem (Kruskal 1960). The criteria requires that for every infinite sequence  $e_1, e_2 \dots$  there exist indices  $i < j$  such that  $e_j \not\prec e_i$ . The intuition of homeomorphic embedding is that  $x \not\prec y$  holds if by removing nodes from  $y$  you cannot obtain  $x$ . Our termination rule uses similar ideas to a well-quasi ordering, but with a very different comparison relation.

We are unaware of any other supercompilers that have assigned names to expressions, or that have used a bag based termination rule (most use tree orderings, or sometimes cost models/budgets). Without our particular treatment of expressions as a set of let bindings, and our particular simplification rules, it is not possible to use our termination rule. For example, if we ever inline let bindings then subexpressions would be changed internally, and a single name for each subexpression would no longer be sufficient.

In some cases, our rule is certainly less restrictive than the homeomorphic embedding. The example in §2.6.4 would have stopped one step earlier with a homeomorphic embedding. Under a fairly standard interpretation of variable names and let expressions, we can show that our rule is always less restrictive than the homeomorphic embedding – although other differences in our treatment of expressions mean such a comparison is not necessarily meaningful. However, we did not choose our termination criteria to permit more expressions – it was chosen for both simplicity and compilation speed.

We use two separate termination histories, one in `reduce` and another in `optimise` – an idea suggested by Mitchell (2008), but not previously implemented. By separating the termination histories we gain better predictability, as `reduce` is not dependent on which functions have gone before. Additionally, the histories are kept substantially smaller, again improving compile-time performance. By splitting termination checks we also reduce the coupling between the separate aspects of supercompilation, allowing us to present a simpler manager than would otherwise be possible.

As a result of the changes to termination and the Core language, the operation for splitting when the termination check fails is radically different. In particular, we can use almost identical operations when either evaluation fails to continue, or the termination check fails.

## 5.2 Partial evaluation

There has been a lot of work on partial evaluation (Jones et al. 1993), where a program is specialised with respect to some static data. Partial evaluation works by marking all variable bindings within a program as either static or dynamic, using binding time analysis, then specialises the program with respect to the static bindings. Partial evaluation is particularly appropriate for optimising an interpreter with respect to the expression tree of a particular program, automatically generating a compiler, and removing *interpretation overhead*. The translation of an interpreter into a compiler is known as the First Futamura Projection (Futamura 1999), and can often give an order of magnitude speedup.

Supercompilation and partial evaluation both remove abstraction overhead within a program. Partial evaluation is more suited to completely removing static data, such as an expression tree which is interpreted. Supercompilation is able to remove intermediate data structures, which partial evaluation cannot usually do.

## 5.3 Deforestation

Deforestation (Wadler 1990) removes intermediate trees (most commonly lists) from computations. This technique has been extended in many ways, including to encompass higher-order deforestation (Marlow 1996). In many cases the gains from supercompilation are just particular forms of deforestation.

Probably the most practically applied work on deforestation uses GHC’s rewrite rules to optimise programs (Peyton Jones et al. 2001). Shortcut deforestation rewrites many definitions in terms of `foldr` and `build`, then combines `foldr/build` pairs (Gill et al. 1993) to deforest lists. Stream fusion works similarly, but relies on `stream/unstream` rules (Coutts et al. 2007a). All these schemes are only able to optimise functions written in terms of the correct primitives, which have had fusion rules defined. The advantage of supercompilation is that it applies to many types and functions, without any special effort from the program author.

## 5.4 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 (Peyton Jones and Launchbury 1991). 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 (Peyton Jones 1992), 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 (Marlow et al. 2007). We rely on GHC to perform all these optimisations after our supercompiler generates a target program.

The GRIN approach (Boquist and Johnsson 1996) uses whole program compilation for Haskell. It is currently being implemented

in the `jhc` compiler (Meacham 2008), with promising initial results. GRIN works by first removing all functional values, turning them into case expressions, allowing subsequent optimisation. 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.

## 6. Conclusions and Future Work

We have described a novel supercompiler, with a focus on simplicity, which can compile our benchmarks in a few seconds, and in some benchmarks offers substantial performance improvements over GHC alone. We see two main avenues for future work: increasing the range of benchmarks, and improving the runtime performance.

### 6.1 More Benchmarks

In order to run more benchmarks we need to automatically translate Haskell to our Core language. In previous papers we used the `Yhc` compiler to generate Core (Golubovsky et al. 2007), but sadly `Yhc` is not maintained and no longer works. Given that our supercompiler relies on GHC to perform strictness analysis and generate native code, it seems sensible to use GHC to generate our Core language – perhaps as a compiler plug-in, or working on external Core, or integrated into the compiler.

Our supercompiler processes the whole program in one go, which naturally leads to questions of scalability. In the tests we have run we have not had a problem with compilation time, but it is something to be aware of as benchmarks increase in size. We believe that our supercompiler could be sped up massively, using some of the techniques mentioned in §4.2. In addition, we could split programs into separate components by defining some functions to be primitive – although this will remove optimisation potential.

### 6.2 Runtime Performance

Our performance results are good, but there are always opportunities to improve. We currently rely on GHC’s strictness analysis to run after we have optimised the program, but by integrating a strictness analysis we may be able to do better. The most common uses of GHC’s rules engine, particularly `list/stream fusion`, are automatically performed by our supercompiler. However, some transformations such as replacing `head ∘ sort` with `minimum`, are too complex to automatically infer. It may be of benefit to integrate a rules engine in to our supercompiler.

In some cases the author of a program has a particular idea about some intermediate data structure they expect to be eliminated. If these structures remain in the optimised program the performance penalty is sometimes dramatic. Perhaps a user could mark some values they expect to be removed, and then be warned if they remain.

### 6.3 Conclusions

Supercompilation is a powerful technique which generalises many of the transformations performed by optimising compilers. We were initially drawn to supercompilation for two reasons. Firstly, all intermediate values have the potential to be eliminated, regardless of their type or the functions which operate on them. Secondly, supercompilation is a single-pass optimisation, avoiding the tricky problem of ordering compiler phases for best optimisation. With these advantages supercompilation has the potential to drastically simplify an optimising compiler, while still achieving great performance. Our supercompiler builds on these advantages, rethinking supercompilation to make it simpler and improve compilation times.

## Acknowledgements

I would like to thank Max Bolingbroke, Jason Reich, Simon Peyton Jones and Peter Jonsson for helpful discussions. Thanks to Ketil Malde for providing further inspiration to continue researching supercompilation. Thanks to Max Bolingbroke, Mike Dodds and the anonymous referees for helpful comments on earlier drafts.

## References

- 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.
- Olaf Chitil. Common subexpressions are uncommon in lazy functional languages. *LNCS*, 1467:53–71, 1998.
- 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 2007a.
- Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Proc PADL 2007*, pages 50–64. Springer-Verlag, January 2007b.
- Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc PDLI '93*, volume 28(6), pages 237–247. ACM Press, New York, 1993.
- Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
- Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proc FPCA '93*, pages 223–232. ACM Press, June 1993.
- Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core – from Haskell to Core. *The Monad.Reader*, 1(7): 45–61, April 2007.
- Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. FPCA '85*, pages 190–203. Springer-Verlag, 1985.
- Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
- Peter Jonsson and Johan Nordlander. Positive supercompilation for a higher order call-by-value language. In *POPL '09*, pages 277–288. ACM, 2009.
- Ilya Klyuchnikov. Supercompiler HOSC 1.0: under the hood. In *KIAM*, volume preprint no. 63, 2009.
- Ilya Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. In *KIAM*, volume preprint no. 21, 2010.
- J Kort. Deforestation of a raytracer. Master's thesis, University of Amsterdam, 1996.
- Joseph Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–255, 1960.
- Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The essence of computation: complexity, analysis, transformation*, pages 379–403. Springer-Verlag, 2002.
- Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, 1996.
- 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.
- John Meacham. jhc: John's Haskell compiler. <http://repetae.net/john/computer/jhc/>, 2008.
- Neil Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, 2008.
- Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In *Selected papers from IFL 2007*, volume 5083 of *LNCS*, pages 147–164. Springer-Verlag, May 2008.
- Will Partain et al. The nofib Benchmark Suite of Haskell Programs. <http://darcs.haskell.org/nofib/>, 2008.
- Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *JFP*, 2(2): 127–202, 1992.
- Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proc. ICFP '07*, pages 327–337. ACM Press, October 2007.
- Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proc FPCA '91*, volume 523 of *LNCS*, pages 636–666. Springer-Verlag, August 1991.
- Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12:393–434, July 2002.
- 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.
- 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.
- Jens Peter Secher and Morten Sørensen. On perfect supercompilation. In *Proceedings of Perspectives of System Informatics*, volume 1755 of *LNCS*, pages 113–127. Springer-Verlag, 2000.
- Morten Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
- The GHC Team. The GHC compiler, version 6.12.1. <http://www.haskell.org/ghc/>, December 2009.
- Andrew Tolmach. An external representation for the GHC core language. <http://www.haskell.org/ghc/docs/papers/core.ps.gz>, September 2001.
- Valentin Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.
- Valentin Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Computation*, pages 341–353. North-Holland, 1988.
- Valentin Turchin. *Refal-5, Programming Guide & Reference Manual*. New England Publishing Co., Holyoke, MA, 1989.
- Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
- Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76. ACM Press, 1989.