Supercompilation by Evaluation

Max Bolingbroke
University of Cambridge
mb566@cam.ac.uk

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

Supercompilation is a technique due to Turchin [1] which allows for the construction of program optimisers that are both simple and extremely powerful. Supercompilation is capable of achieving transformations such as deforestation [2], function specialisation and constructor specialisation [3]. Inspired by Mitchell's promising results [4], we show how the call-by-need supercompilation algorithm can be recast to be based explicitly on an evaluator, and in the process extend it to deal with recursive $\mathbf{let}$ expressions.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory – Semantics; D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors – Optimization

General Terms Supercompilation, Performance

1. Overview

The key contributions of this paper are as follows:

- We cast supercompilation in a new light, showing how to design a modular supercompiler that is based *directly* on the operational semantics of the language (Section 3). Viewing supercompilation in this way is valuable, because it makes it easier to derive a supercompiler in a systematic way from the language, and to adapt it to new language features. Previous work intermingles evaluation and specialisation in a much more complex and ad-hoc way.
- As an example of this flexibility, we show how to supercompile a call-by-need language with unrestricted recursive $\mathbf{let}$ bindings, by making use of a standard evaluator for call-by-need (Section 4). This has two advantages:
  - Our supercompiler can deforest the following term:

    $\mathbf{let}\ ones = 1 : ones; map = \ldots$
    $\mathbf{in}\ map\ (\lambda x.\, x + 1)\ ones$

    into the direct-style definition:

    $\mathbf{let}\ xs = 2 : xs\ \mathbf{in}\ xs$

    No other existing supercompiler achieves this, to our knowledge; previous supercompilers for lazy languages have dealt only with non-recursive $\mathbf{let}$ bindings.
  - Because recursion is not special, we do not need to give the program top-level special status, or λ-lift the input program.
- We perform an empirical evaluation of our supercompiler (Section 5), in particular comparing it to Mitchell's supercompiler [4]. Our supercompiler reduces benchmark runtime by up to 95%, with a harmonic mean reduction of 70%.

2. Supercompilation by example

The best way to understand how supercompilation works is by example. Let's begin with a simple example of how standard supercompilation can specialise functions to their higher-order arguments:

$$\begin{array}{l} \mathbf{let}\ inc = \lambda x.\, x + 1 \\ \quad map = \lambda f\ xs.\, \mathbf{case}\ xs\ \mathbf{of}\ [\,] \to [\,] \\ \qquad\qquad\qquad\qquad\qquad (y : ys) \to f\ y : map\ f\ ys \\ \mathbf{in}\ map\ inc\ zs \end{array}$$

A supercompiler evaluates open terms, so that reductions that would otherwise be done at runtime are performed at compile time. Consequently, the first step of the algorithm is to reduce the term as much as possible, following standard evaluation rules:

$$\begin{array}{l} \mathbf{let}\ inc = \ldots; map = \ldots \\ \mathbf{in}\ \mathbf{case}\ zs\ \mathbf{of}\ [\,] \to [\,] \\ \qquad\qquad\qquad (y : ys) \to inc\ y : map\ inc\ ys \end{array}$$

At this point, we become stuck on the free variable zs. The most important decision when designing a supercompiler is how to proceed in such a situation, and we will spend considerable time later explaining how this choice is made when we cover the *splitter* in Section 3.5. In this particular example, we continue by recursively supercompiling two subexpressions. We intend to later recombine the two subexpressions into an output term where the $\mathbf{case}\ zs$ remains in the output program, but where both branches of the case have been further optimised by supercompilation.

The first subexpression is just $[\,]$. Because this is already a value, supercompilation makes no progress: the result of supercompiling that term is therefore $[\,]$.

The second subexpression is:

$$\begin{array}{l} \mathbf{let}\ inc = \ldots; map = \ldots \\ \mathbf{in}\ inc\ y : map\ inc\ ys \end{array}$$

Again, evaluation of this term is unable to make progress: the rules of call-by-need reduction do not make allowance for evaluating within non-strict contexts such as the arguments of data constructors. It is once again time to use the splitter to produce some subexpressions suitable for further supercompilation.

This time, the first subexpression is:

$$\mathbf{let}\ inc = \ldots\ \mathbf{in}\ inc\ y$$

Again, we perform reduction, yielding the supercompiled term $y + 1$. The other subexpression, originating from splitting the $(y : ys)$ **case** branch, is:

$$\begin{array}{l}\mathbf{let}\ inc = \ldots ; map = \ldots \\ \mathbf{in}\ map\ inc\ ys\end{array}$$

This term identical to the one we started with, except that it has the free variable ys rather than zs. If we continued inlining and β-reducing the map call, the supercompiler would not terminate. This is not what we do.

Instead, the supercompiler uses a *memo function*. It records all of the terms it has been asked to supercompile as it proceeds, so that it never supercompiles the same term twice. In concrete terms, it builds up a set of *promises*, each of which is an association between a term previously submitted for supercompilation, its free variables, and a unique, fresh name (typically written $h0$, $h1$, etc.). At this point in the supercompilation of our example, the promises will look something like this:

$$\begin{array}{ll}h0\ zs & \mapsto \mathbf{let}\ inc = \ldots ; map = \ldots \mathbf{in}\ map\ inc\ zs \\ h1 & \mapsto [\,] \\ h2\ y\ ys & \mapsto \mathbf{let}\ inc = \ldots ; map = \ldots \mathbf{in}\ inc\ y : map\ inc\ ys \\ h3\ y & \mapsto \mathbf{let}\ inc = \ldots \mathbf{in}\ inc\ y\end{array}$$

We have presented the promises in a rather suggestive manner, as if the promises were a sequence of bindings. Indeed, the intention is that the final output of the supercompilation process will be not only an optimised expression, but one optimised binding for each $h0$, $h1$, ... ever added to the promises.

Because the term we are now being asked to supercompile is is simply a renaming of the original term (with which we associated the name $h0$) we can immediately return $h0\ ys$ as the supercompiled version of the current term. Producing a *tieback* like this we can rely on the (not yet known) optimised form of the original term (rather than supercompiling afresh), while simultaneously sidestepping a possible source of non-termination.

Now, both of the recursive supercompilations requested in the process of supercompiling $h1$ have been completed. We can now rebuild the optimised version of the $h2$ term from the optimised subterms, which yields:

$$h3\ y : h0\ ys$$

Continuing this process of rebuilding an optimised version of the supercompiler input from the optimised subexpressions, we eventually obtain this final program:

$$\begin{array}{l}\mathbf{let}\ h0\ zs = \mathbf{case}\ zs\ \mathbf{of}\ [\,] \to h1 ; (y : ys) \to h2\ y\ ys \\ \quad h1 = [\,] \\ \quad h2\ y\ ys = h3\ y : h0\ ys \\ \quad h3\ y = y + 1 \\ \mathbf{in}\ h0\ zs\end{array}$$

A trivial post-pass can eliminate some of the unnecessary indirections to obtain a version of the original input expression, where map has been specialised on its functional argument:

$$\begin{array}{l}\mathbf{let}\ h0\ zs = \mathbf{case}\ zs\ \mathbf{of}\ [\,] \to [\,] ; (y : ys) \to (y + 1) : h0\ ys \\ \mathbf{in}\ h0\ zs\end{array}$$

3. The basic supercompiler

We now describe the design of an unusually-modular supercompiler for a simple functional language that closely approximates GHC's intermediate language, Core. The syntax of the language itself in presented in Figure 1; it is a standard untyped call-by-need calculus with recursive **let**, algebraic data types, primitive literals and strict primitive operations. Although Figure 1 describes terms

Variables x, y, z **Primitives** $\otimes ::= +, -, \ldots$

Data Constructors $\mathbf{C} ::= True, Just, (:), \ldots$

Literals $\ell ::= 1, 2, \ldots, \texttt{'a'}, \texttt{'b'}, \ldots$

Values

v	$::=$	$\lambda x.\, e$	Lambda abstraction
	$\mid$	ℓ	Literal
	$\mid$	$\mathbf{C}\ \overline{x}$	Saturated constructed data

Terms

e	$::=$	x	Variable reference
	$\mid$	v	Values
	$\mid$	$e\ x$	Application
	$\mid$	$e \otimes e$	Binary primops
	$\mid$	$\mathbf{let}\ \overline{x = e}\ \mathbf{in}\ e$	Recursive let-binding
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{\alpha \to e}$	Case decomposition

Case Alternative

α	$::=$	ℓ	Literal alternative
	$\mid$	$\mathbf{C}\ \overline{x}$	Constructor alternative

Heaps $H ::= \overline{x \mapsto e}$ **Stacks** $K ::= \overline{\kappa}$

Stack Frames

κ	$::=$	$\mathbf{update}\ x$	Update frame
	$\mid$	$\bullet\ x$	Apply to function value
	$\mid$	$\mathbf{case} \bullet \mathbf{of}\ \overline{\alpha \to e}$	Scrutinise value
	$\mid$	$\bullet \otimes e$	Apply first value to primop
	$\mid$	$v \otimes \bullet$	Apply second value to primop

Figure 1: Syntax of the Core language and evaluator

in A-normal form [5], for clarity of presentation we will often write non-normalised expressions. A program is simply a term, in which the top-level function definitions appear as possibly-recursive let bindings.

A small-step operational semantics of Core appears in Figure 3, and is completely conventional in the style of Sestoft [6] — so conventional that our description here is very brief indeed. The state of the machine is a triple $\langle H \mid e \mid K \rangle$, of a heap, a term and a stack. The term is the *focus* of evaluation, while the stack embodies the evaluation context, or continuation, that will consume the value produced by the term. Figure 1 gives the syntax of heaps and stacks, as well as terms.

Our supercompiler is built from the following four, mostly independent, subsystems:

1. A *termination criteria* that prevents the supercompiler from running forever: Section 3.2
2. An *evaluator* for the language under consideration: Section 3.3
3. A *memoiser*, which ensures that we supercompile any term at most once: Section 3.4
4. A *splitter* that tells us how to proceed when evaluation becomes blocked: Section 3.5

We will show how to implement each of these components in a way that will yield a standard supercompiler, which is nonetheless more powerful than previous work in that it will naturally support recursive **let**.

$$
\begin{array}{l}
\textbf{type } \mathit{Heap} = \mathit{Map}\ \mathit{Var}\ \mathit{Term} \\
\textbf{type } \mathit{Stack} = [\mathit{StackFrame}] \\
\textbf{data } \mathit{StackFrame} = \ldots \quad \text{-- See } \kappa \text{ in Figure 1} \\
\textbf{data } \mathit{Term} = \ldots \quad \text{-- See } e \text{ in Figure 1} \\
\textbf{type } \mathit{State} = (\mathit{Heap}, \mathit{Term}, \mathit{Stack}) \\
\mathit{freeVars} :: \mathit{State} \to [\mathit{Var}] \\
\mathit{rebuild} :: \mathit{State} \to \mathit{Term} \\
\mathit{sc} :: \mathit{History} \to \mathit{State} \to \mathit{ScpM}\ \mathit{Term} \\
\quad \text{-- The evaluator (Section 3.3)} \\
\mathit{reduce} :: \mathit{State} \to \mathit{State} \\
\quad \text{-- The splitter (Section 3.5)} \\
\mathit{split} :: \mathit{Monad}\ m \Rightarrow (\mathit{State} \to m\ \mathit{Term}) \\
\qquad\qquad\qquad \to \mathit{State} \to m\ \mathit{Term} \\
\quad \text{-- Termination checking (Section 3.2)} \\
\textbf{type } \mathit{History} = [\mathit{State}] \\
\mathit{emptyHistory} = [\,] :: \mathit{History} \\
\mathit{terminate} :: \mathit{History} \to \mathit{State} \to \mathit{TermRes} \\
\textbf{data } \mathit{TermRes} = \mathit{Stop} \mid \mathit{Continue}\ \mathit{History} \\
\quad \text{-- Memoisation and the ScpM monad (Section 3.4)} \\
\mathit{memo} :: (\mathit{State} \to \mathit{ScpM}\ \mathit{Term}) \\
\qquad \to \mathit{State} \to \mathit{ScpM}\ \mathit{Term} \\
\mathit{match} :: \mathit{State} \to \mathit{State} \to \mathit{Maybe}\ (\mathit{Var} \to \mathit{Var}) \\
\mathit{runScpM}\ \ :: \mathit{ScpM}\ \mathit{Term} \to \mathit{Term} \\
\mathit{freshName} :: \mathit{ScpM}\ \mathit{Var} \\
\mathit{promises}\ \ :: \mathit{ScpM}\ [\mathit{Promise}] \\
\mathit{promise}\ \ \ :: \mathit{Promise} \to \mathit{ScpM}\ () \\
\mathit{bind}\ \ \ \ \ :: \mathit{Var} \to \mathit{Term} \to \mathit{ScpM}\ () \\
\textbf{data } \mathit{Promise} = P\ \{\mathit{name} :: \mathit{Var}, \\
\qquad\qquad\qquad\qquad \mathit{fvs} :: [\mathit{Var}], \\
\qquad\qquad\qquad\qquad \mathit{meaning} :: \mathit{State}\}
\end{array}
$$

Figure 2: Types used in the standard supercompiler

3.1 The top-level

A distinctive feature of our supercompiler is that it operates on States rather than Terms; we reflect on why in Section 3.7. A State is a triple of type $(\mathit{Heap}, \mathit{Term}, \mathit{Stack})$, and it represents precisely the state $\langle H \mid e \mid K \rangle$ of the abstract machine (Figure 3).

Notice that Term and State are isomorphic: any Term can be converted to its initial State, and any State can be converted back to a Term simply by wrapping the heap and the stack around the term, a function we call $\mathit{rebuild}$. The signatures of the major functions and data types used by the supercompiler – including State and $\mathit{rebuild}$ – are given for easy reference in Figure 2.

The core of the supercompilation algorithm is sc, whose key property is this: for any history h and state s, $(\mathit{sc}\ h\ s)$ returns a term with exactly the same meaning as s, but which is implemented more efficiently.

$$
\begin{array}{l}
\mathit{sc}, \mathit{sc}' :: \mathit{History} \to \mathit{State} \to \mathit{ScpM}\ \mathit{Term} \\
\mathit{sc}\ \mathit{hist} = \mathit{memo}\ (\mathit{sc}'\ \mathit{hist}) \\
\mathit{sc}'\ \mathit{hist}\ \mathit{state} = \textbf{case } \mathit{terminate}\ \mathit{hist}\ \mathit{state}\ \textbf{of} \\
\quad \mathit{Continue}\ \mathit{hist}' \to \mathit{split}\ (\mathit{sc}\ \mathit{hist}')\ (\mathit{reduce}\ \mathit{state}) \\
\quad \mathit{Stop} \qquad\qquad\ \to \mathit{split}\ (\mathit{sc}\ \mathit{hist})\ \ \mathit{state}
\end{array}
$$

As foreshadowed in Section 2, sc is a memoised function: if it is ever asked to supercompile a State that is identical to one we have previously supercompiled (modulo renaming), we want to reuse that previous work. This is achieved by calling memo, which memoises uses of sc by recording information in the ScpM monad. We will describe memoisation in more detail in Section 3.4.

Memoisation deals with the case where sc is called on an identical argument. But what if it is called on a *growing* argument? You might imagine that we would keep supercompiling forever. This well-known problem arises, for example, when supercompiling a recursive function with an accumulating parameter.

There is likewise well-known way to ensure that supercompilation terminates, which involves maintaining a "history" of previous arguments. In concrete terms, the parameter hist is the history, and sc' starts by calling $\mathit{terminate}$ (Figure 2) to decide whether to Stop or (the common case) $\mathit{Continue}$. The implementation of histories and $\mathit{terminate}$ is elaborated in Section 3.6. The normal case is that $\mathit{terminate}$ returns $\mathit{Continue}\ \mathit{hist}'$, in which case sc' proceeds thus:

1. It invokes a call-by-need evaluator, reduce, to optimise the state s by evaluating it to head normal form. This amounts to performing compile-time evaluation, so reduce must itself be careful not to diverge – see Section 3.3.
2. It uses split to recursively supercompile some subcomponents of the reduced state, optimising parts of the term that reduction didn't reach.

Here is an example. Imagine that this term was input to sc[1]:

$$
\begin{array}{l}
\textbf{let } x = \mathit{True}; y = 1 + 2 \\
\textbf{in case } x \textbf{ of } \mathit{True} \to \mathit{Just}\ y; \mathit{False} \to \mathit{Nothing}
\end{array}
$$

Assuming that this State has never been previously supercompiled, sc' will be invoked by memo. Further assuming that the termination check in sc' returns $\mathit{Continue}$, we would reduce the input state to head normal form, giving a new state':

$$\textbf{let } y = 1 + 2 \textbf{ in } \mathit{Just}\ y$$

The **case** computation and x binding have been reduced away. It would be possible to return this state' as the final, supercompiled form of our input — indeed, in general the supercompiler is free to stop at any time, using $\mathit{rebuild}$ to construct a semantically-equivalent result term. However, doing so misses the opportunity to supercompile some *subcomponents* of state' that are not reduced in the head normal form. Instead, we feed state' to split, which:

1. Invokes $\mathit{sc}\ \mathit{hist}'$ on the subterm $1 + 2$, achieving further supercompilation (and hence optimisation). Let's say for the purposes of the example that this then returns the final optimised term $\mathit{h1}$, with a corresponding optimised binding $\mathit{h1} = 3$ recorded in the monad.
2. Reconstructs the term using the optimised subexpressions. So in this case the Term returned by split would be $\textbf{let } y = \mathit{h1} \textbf{ in } \mathit{Just}\ y$.

The entry point to the supercompiler, start, is as follows:

$$
\begin{array}{l}
\mathit{start} :: \mathit{Term} \to \mathit{Term} \\
\mathit{start}\ e = \mathit{runScpM}\ (\mathit{sc}\ \mathit{emptyHistory}\ (\mathit{emptyHeap}, e, [\,]))
\end{array}
$$

The input term, e, is first converted into an initial State, namely $(\mathit{emptyHeap}, e, [\,])$. This initial state is passed to the main supercompiler sc, along with the initial history. Finally sc is performed in the ScpM monad, initialised by $\mathit{runScpM}$ – we describe this monad in detail in Section 3.4.

In the following sections, we will explore the meaning and implementation of the reduce, memo, $\mathit{terminate}$ and split functions in much more detail.

[1] Technically sc takes a State not a Term, but in our examples we will often write a term e in place of the state $(\mathit{emptyHeap}, e, [\,])$, as we do here.

3.2 The termination criteria

The core of the supercompiler's termination check is provided by a single function, $terminate$:

$$\begin{array}{l} terminate :: History \to State \to TermRes \\ \mathbf{data}\ TermRes = Stop \mid Continue\ History \end{array}$$

As the supercompiler proceeds, it builds up an ever-larger $History$ of previously-observed $State$s. This history is both interrogated and extended by calling $terminate$. Termination is guaranteed by making sure that $History$ cannot grow indefinitely.

More precisely, $terminate$ guarantees that, for any history h_0 and states $s_0, s_1, s_2, \ldots$ there can be no infinite sequence of calls to $terminate$ of this form:

$$\begin{array}{rcl} terminate\ h_0\ s_0 & = & Continue\ h_1 \\ terminate\ h_1\ s_1 & = & Continue\ h_2 \\ \ldots & & \\ terminate\ h_i\ s_i & = & Continue\ h_{i+1} \end{array}$$

Instead, there will always exist some j such that:

$$terminate\ h_j\ s_j\ =\ Stop$$

In Section 3.3 we will see how $reduce$ uses $terminate$ to ensure that it only performs a bounded number of reduction steps, and we will discuss how $terminate$ ensures that the overall supercompiler terminates in Section 3.6.

So much for the specification, but how can $terminate$ be implemented? Of course, $(\lambda xy.\ Stop)$ would be a sound implementation of $terminate$, in that it satisfies the property described above, but it is wildly over-conservative because it forces the supercompiler to stop reduction *immediately*. We want an implementation of $terminate$ that is correct, but which nonetheless waits for as long as possible before preventing further reduction by answering $Stop$.

The key to implementing such a termination criteria is defining a *well-quasi-order* [7]. The relation $\lhd \in S \times S$ is a well-quasi-order iff for all infinite sequences of elements of S $(s_0, s_1, \ldots)$, there $\exists ij. i < j \wedge s_i \lhd s_j$. Given any well-quasi-order $\lhd : State \times State$, we can implement a correct $terminate$ function:

$$\begin{array}{l} terminate\ prevs\ here \\ \quad = \mathbf{if}\ any\ (\lhd here)\ prevs\ \mathbf{then}\ Stop \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ Continue\ (here : prevs) \end{array}$$

Concretely, we choose the tag-bag ordering of Mitchell [4] as the basis of our well-quasi-order. The tag-bag order relates bags (multisets) of "tags" as follows:

$$t_1 \lhd_{tb} t_2 \iff set\,(t_1) = set\,(t_2) \wedge |t_1| \leq |t_2|$$

For this to be a well-quasi-order there must be a finite number of distinct tags that can appear in the bags. We take tags to be Ints, and assume that every sub-term of the supercompiler's input program is labelled with a unique Int, which forms the tag for that expression. Likewise, $StackFrame$s are labelled with the tag of the term the evaluator produced them from – e.g. a **case** $\bullet$ **of**$\overline{\alpha\ \to\ e}$ frame would be labelled with the tag of the corresponding **case** expression. Occasionally, the evaluator needs to manufacture a new term which did not necessarily occur in the input program – e.g. if we evaluate $1 + 2$ to get the new value 3. In such cases, one of the operand tags is used as the tag for the new term.

The termination criteria then defines an internal function that obtains a tag-bag from the components of a $State$ triple:

$$\begin{array}{l} tagBag :: State \to Bag\ Tag \\ tagBag\ (h, e, k) \\ \quad = (termTag\ e * 2)\ \text{`}insertBag\text{`} \\ \qquad fmap\ (*3)\ (heapTagBag\ h)\ \text{`}plusBag\text{`} \\ \qquad fmap\ (*5)\ (stackTagBag\ k)\ \text{`}plusBag\text{`} \end{array}$$

The $tagBag$ function multiplies tags by distinct prime numbers depending on where in the evaluation context the tag originated from. This does not change the fact that there are only ever a finite number of distinct tags in the bags (and hence $\lhd_{tb}$ is still a well-quasi-order). However, the multiplication prevents the evaluator from terminating just because e.g. a tagged binding that used to appear in the $Heap$ is forced and hence has its tag show up on a $StackFrame$ instead.

Finally, we can combine $tagBag$ and $\lhd_{tb}$ to produce the well-quasi-order $\lhd$ on $State$s used by $terminate$:

$$\begin{array}{l} (\lhd) :: State \to State \to Bool \\ s1 \lhd s1 = tagBag\ s1 \lhd_{tb} tagBag\ s2 \end{array}$$

Mitchell uses tag-bags in a similar way, but only associates tags with let-bound variables. In order to tag every subexpression, he keeps terms in a normal form where all subexpressions are let-bound. Supercompiling $State$s and tagging subterms directly means that we can avoid let-floating and – because we distinguish between tags from subexpressions currently being evaluated (in the stack), and those subexpressions that are not in the process of being forced (in the heap) – our termination criteria is more lenient.

3.3 The evaluator

The $reduce$ function tries to reduce a $State$ to head normal form. In case the term diverges, $reduce$ includes a termination check that allows it to stop after a finite number of steps. (This check is conservative, of course, so $reduce$ might fail to find a head normal form when one does exist.) The two key properties of $reduce$ are:

- Reduction preserves meaning: the $State$ returned has the same semantics as the input $State$
- Regardless of what meaning the input $State$ may have, $reduce$ always terminates

The implementation is straightforward:

$$\begin{array}{l} reduce :: State \to State \\ reduce = go\ emptyHistory \\ \mathbf{where} \\ \quad go\ hist\ state = \mathbf{case}\ step\ state\ \mathbf{of} \\ \qquad Nothing \to state \\ \qquad Just\ state' \\ \qquad\quad \mid intermediate\ state' \to go\ hist\ state' \\ \qquad\quad \mid otherwise \to \mathbf{case}\ terminate\ hist\ state'\ \mathbf{of} \\ \qquad\qquad Stop \qquad\quad\ \to state' \\ \qquad\qquad Continue\ hist' \to go\ hist'\ state' \\ \quad intermediate\ (\_, Var\ \_, \_) = False \\ \quad intermediate\ \_ \qquad\qquad\ \ = True \\ step :: State \to Maybe\ State \\ \quad \text{-- Implements Figure 3} \end{array}$$

The $reduce$ function uses a loop, the function go, with an accumulating history. In turn go uses an internal function, $step$, which implements precisely the one-step reduction relation of Figure 3. Note that $step$ returns a $Maybe\ State$ – this accounts for reduction being unable to proceed due to either reaching a value, or because a variable is in the focus which is not bound by the heap (remember that $reduce$ may be used on open terms). In that case $reduce$ terminates with the state it has reached.

The totality of $reduce$ is achieved using the $terminate$ function. If $terminate$ reports that evaluation appears to be diverging, $reduce$ immediately returns. As a result, the $State$ triple (h, e, k) returned by $reduce$ might not be fully reduced – in particular, it might be the case that $e \equiv Var\ x$ where x is bound by h.

As an optimisation, the termination criteria is not tested if the $State$ is considered to be "intermediate". The $intermediate$ pred-

$$\boxed{\langle H \mid e \mid K\rangle \rightsquigarrow \langle H \mid e \mid K\rangle}$$

VAR	$\langle H, x \mapsto e \mid x \mid K\rangle$	$\rightsquigarrow$	$\langle H \mid e \mid \textbf{update}\ x, K\rangle$
UPDATE	$\langle H \mid v \mid \textbf{update}\ x, K\rangle$	$\rightsquigarrow$	$\langle H, x \mapsto v \mid v \mid K\rangle$
APP	$\langle H \mid e\ x \mid K\rangle$	$\rightsquigarrow$	$\langle H \mid e \mid \bullet\ x, K\rangle$
LAMBDA	$\langle H \mid \lambda x.\, e \mid \bullet\ x, K\rangle$	$\rightsquigarrow$	$\langle H \mid e \mid K\rangle$
PRIM	$\langle H \mid e_1 \otimes e_2 \mid K\rangle$	$\rightsquigarrow$	$\langle H \mid e_1 \mid \bullet \otimes e_2, K\rangle$
PRIM-LEFT	$\langle H \mid v_1 \mid \bullet \otimes e_2, K\rangle$	$\rightsquigarrow$	$\langle H \mid e_2 \mid v_1 \otimes \bullet, K\rangle$
PRIM-RIGHT	$\langle H \mid v_2 \mid v_1 \otimes \bullet, K\rangle$	$\rightsquigarrow$	$\langle H \mid \otimes (v_1, v_2) \mid K\rangle$
CASE	$\langle H \mid \textbf{case}\ e_{\text{scrut}}\ \textbf{of}\ \overline{\alpha \to e} \mid K\rangle$	$\rightsquigarrow$	$\langle H \mid e_{\text{scrut}} \mid \textbf{case} \bullet \textbf{of} \overline{\alpha \to e}, K\rangle$
DATA	$\langle H \mid \mathbf{C}\ \overline{x} \mid \textbf{case} \bullet \textbf{of} \{\ldots, \mathbf{C}\ \overline{x} \to e, \ldots\}, K\rangle$	$\rightsquigarrow$	$\langle H \mid e \mid K\rangle$
LIT	$\langle H \mid \ell \mid \textbf{case} \bullet \textbf{of} \{\ldots, \ell \to e, \ldots\}, K\rangle$	$\rightsquigarrow$	$\langle H \mid e \mid K\rangle$
LETREC	$\langle H \mid \textbf{let}\ \overline{x = e}\ \textbf{in}\ e_{\text{body}} \mid K\rangle$	$\rightsquigarrow$	$\langle H, \overline{x \mapsto e} \mid e_{\text{body}} \mid K\rangle$

Figure 3: Operational semantics of the Core language

icate shown ensures that we only test for non-termination upon reaching a variable – this is safe because every infinite series of reduction steps must certainly have a variable occur in the focus an infinite number of times. After some experience with our supercompiler we discovered that making termination tests infrequent is actually more than a mere optimisation. If we test for termination very frequently (say, after every tiny step), the successive states will be very similar; and the more similar they are, the greater the danger that the necessarily-conservative termination criterion (Section 3.2) will unnecessarily say $Stop$. (For example, in the limit, it must say $Stop$ for two identical states.)

3.4 The memoiser

The purpose of the memoisation function, $memo$, is to ensure that we never supercompile a term more than once. We achieve this by using the $ScpM$ monad to record information about previously supercompiled $States$. Precisely, the $ScpM$ monad is a simple state monad with three pieces of state:

1. The *promises*, which comprise all the $State$s that have been previously submitted for supercompilation, along with:
   - The names that the supercompiled versions of those $State$s will be bound to in the final program (e.g. $h0$, $h1$)
   - The list of free variables that those bindings will be abstracted over[2]. By instantiating these free variables several different ways, we can reuse the supercompiled version of a $State$ several times.

   The data structure used to store all this information is called a $Promise$ (Figure 2).
2. The *optimised bindings*, each of the form $x = e$. The $runScpM$ function, which is used to actually execute $ScpM\ \ Term$ computations, wraps the optimised bindings collected during the supercompilation process around the final supercompiled $Term$ in order to produce the final output.
3. A supply of fresh names ($h0$, $h1$, ...) to use for the optimised bindings.

When sc begins to supercompile a $State$, it records a promise for that state; when it finishes supercompiling that state it records a corresponding optimised binding for it. At any moment there may be unfulfilled promises that lack a corresponding binding, but every binding has a corresponding promise. Moreover, every promise will *eventually* be fulfilled by an entry appearing in the optimised bindings. Figure 2 summarises the signatures of the functions provided by $ScpM$.

We can now implement $memo$ as follows:

```
memo :: (State → ScpM Term)
     → State → ScpM Term
memo opt state = do
  ps ← promises
  let ress = [ (name p `apps` map rn (fvs p))
             | p ← ps
             , Just rn ← [match (meaning p) state]
             ]
  case ress of
    res : _ → return res
    []      → do
      x ← freshName
      let vs = freeVars state
      promise P {name = x, fvs = vs,
                 meaning = state}
      e ← opt state
      bind x (lambdas vs e)
      return (x `apps` vs)
```

The $memo$ function proceeds as follows:

1. Firstly, it examines all existing $promises$. If the $match$ function reports that some existing promise matches the $State$ we want to supercompile (up to renaming), $memo$ returns a call to the optimised binding corresponding to that existing promise.
2. Assuming no promise matches, $memo$ continues:

   (a) A new promise for this novel $State$ is made, in the form of a new $Promise$ entry. A fresh $name$ of the form hn (for some n) is associated with the $Promise$.

   (b) The state is optimised by calling opt, obtaining an optimised term e.

   (c) A final optimised binding $hn\ =\ \lambda\overline{\text{fvs}\,(s)}\ .\ e$ is recorded using $bind$. This binding will be placed in the output program by $runScpM$.

   (d) Finally, a call to that binding, $hn\ \overline{\text{fvs}\,(s)}$, is returned.

[2] Strictly speaking, bindings with no free variables at all should nonetheless be λ-abstracted over a dummy argument (such as ()). This will prevent us from accidentally introducing space leaks by increasing the garbage-collection lifetime of constant expressions.

The $match$ function is used to compare $State$s:

$$match :: State \to State \to Maybe\ (Var \to Var)$$

The key properties of the $match$ function are that:

- If $match\ s1\ s2 \equiv Just\ rn$ then the meaning of $s2$ is the same as that of $rn(s1)$.
- If $s1$ is syntactically identical to $s2$, modulo renaming, then $isJust\ (match\ s1\ s2)$. This property is necessary for termination of the supercompiler, as we will discuss later.

Naturally, it is desirable for the $match$ function to match as many truly equivalent terms as possible. This is made slightly more convenient by the fact that we consider matching $State$s, as they may have already been weakly normalised by the evaluator. Our implementation exploits this by providing a $match$ function that is insensitive to the exact order of bindings in the $Heap$.

One subtle point is that the matching should be careful not to duplicate work. This can happen if an old term such as:

$$\textbf{let}\ x = fact\ 100; y = fact\ 100\ \textbf{in}\ (x, y)$$

is matched against a proposed new one such as:

$$\textbf{let}\ x = fact\ 100\ \textbf{in}\ (x, x)$$

However, if the **let**-bindings in those terms had bound, say, $True$ instead of $fact\ 100$ then matching them would be both permissible and desirable.

3.5 The splitter

The job of the splitter is to somehow continue the process of supercompiling a $State$ which we may not reduce further, either because of a lack of information (e.g. if the $State$ is blocked on a free variable), or because the termination criteria is preventing us from making any further one-step reductions. The splitter has the following type signature:

$$\begin{aligned} split :: Monad\ m \Rightarrow\ & (State \to m\ Term) \\ & \to State \to m\ Term \end{aligned}$$

In general, $(split\ \ opt\ \ s)$ identifies some sub-components of the state s, uses opt to optimise them, and combines the results into a term whose whose meaning is the same as s (assuming, of course, that opt preserves meaning).

A sound, but feeble, implementation of $split\ \ opt\ \ s$ would be one which *never* recursively invokes opt:

$$split\ \_\ s = return\ (rebuild\ s)$$

Such an implementation is wildly conservative, because not even trivially reducible subexpressions will benefit from supercompilation. A good $split$ function will residualise as little of the input as possible, using opt to optimise as much as possible. It turns out that, starting from this sound-but-feeble baseline, there is a rich variety of choices one can make for $split$, as we explore in the rest of this section.

In preparation for describing $split$ in more detail, we first introduce a notational device similar to that of Mitchell [4] for describing the operation of $split$ on particular examples. Suppose that the following $State$ is given to $split$:

$$\langle x \mapsto 1, xs \mapsto map\ (const\ 1)\ ys \mid x : xs \mid \epsilon\rangle$$

In our notation the output of $split$ would be this "term", which has sub-components that are $State$s:

$$\begin{aligned} & \textbf{let}\ x = \langle \epsilon \mid 1 \mid \epsilon\rangle\,; xs = \langle \epsilon \mid map\ (const\ 1)\ ys \mid \epsilon\rangle \\ & \textbf{in}\ x : xs \end{aligned}$$

You should read this in the following way:

- The part of the term outside the ⟨state brackets⟩ is the *residual* code that will form part of the output program.
- In contrast, those things that live within the brackets are the not-yet-residual $State$s which are fed to opt for further supercompilation.

Before $split$ returns, the supercompiled form of the bracketed expressions is pasted into the correct position in the residual code. So the actual end result of such a supercompilation run might be something like:

$$\textbf{let}\ x = h2; xs = h3\ ys\ \textbf{in}\ x : xs$$

where $h2$ and $h3$ will have optimised bindings in the output program, as usual.

So far, we have only seen examples where $split\ \ opt$ invokes opt on subterms of the original input. While this is a good approximation to what $split$ does, in general, we will also want to include some of the context in which that subterm lives. Consider the following input:

$$\langle x \mapsto 1, y \mapsto x + x \mid Just\ y \mid \epsilon\rangle$$

A good way to $split$ is as follows:

$$\textbf{let}\ y = \langle x \mapsto 1 \mid x + x \mid \epsilon\rangle\ \textbf{in}\ Just\ y$$

Note that $split\ \ opt$ decided to recursively optimise the term $x + x$, along with a heap binding for x taken from the context which the subterm lived in. This extra context will allow the supercompiler to reduce $x + x$ to 2 at compile time.

Another way that a subterm can get some context added to it by $split$ when evaluation of a **case** expression gets stuck. As an example, consider the following (stuck) input to $split$:

$$\langle \epsilon \mid x \mid \textbf{case} \bullet \textbf{of}\ (True \to 1; False \to 2)\,, \bullet + 3\rangle$$

One possibility is that $split$ could break the expression up for further supercompilation as follows:

$$\begin{aligned} (\textbf{case}\ x\ \textbf{of}\ \ & True\ \to \langle \epsilon \mid 1 \mid \epsilon\rangle \\ & False \to \langle \epsilon \mid 2 \mid \epsilon\rangle) + \langle \epsilon \mid 3 \mid \epsilon\rangle \end{aligned}$$

However, $split$ can achieve rather more potential for reduction if it duplicates the stack frame performing addition into both **case** branches: in particular, that will mean that we are able to evaluate the addition at compile time:

$$\begin{aligned} (\textbf{case}\ x\ \textbf{of}\ \ & True \to \langle \epsilon \mid 1 \mid (\bullet + 3)\rangle \\ & False \to \langle \epsilon \mid 2 \mid (\bullet + 3)\rangle) \end{aligned}$$

In fact, in general we will always want to push *all* of the stack frames following a $\textbf{case} \bullet \textbf{of}\overline{\alpha \to e}$ frame to meet with the expressions $\overline{e}$ in the **case** branches.

This is one of the places where the decision to have the supercompiler work with $State$s rather than $Term$s pays off: the fact that we have an explicit evaluation context makes the process of splitting at a residual **case** very systematic and easy to implement.

The key property of $split$ is that for any opt that is meaning preserving (such that $opt\ \ s$ returns an expression e with the same meaning as s), $split\ \ opt$ must be meaning preserving in the same sense.

There are a number of subtle points to bear in mind when implementing $split$. We describe some issues below, and will have more to say in Section 4.

Issue 1: learning from residual case branches We gain information about a free variable when it is scrutinised by a residual **case**. Thus, when we have:

$$\langle \epsilon \mid x \mid \textbf{case} \bullet \textbf{of}\ (3 \to x + x; 4 \to x * x)\rangle$$

We split as follows:

$$\begin{aligned}&\textbf{case}\ x\ \textbf{of}\ 3 \to \langle x \mapsto 3 \mid x + x \mid \epsilon\rangle\\ &\qquad\qquad\ 4 \to \langle x \mapsto 4 \mid x * x \mid \epsilon\rangle\end{aligned}$$

Because we have learnt the value of x from the **case** alternative, we are able to statically reduce the $+$ and $*$ operations in each branch.

Issue 2: work duplication Consider splitting the following $State$, where $fact$ is an unknown function and hence must be assumed to to be expensive to execute:

$$\langle x \mapsto fact\ n \mid (x + 1, x + 2) \mid \epsilon\rangle$$

One possibility is to split as follows:

$$(\langle x \mapsto fact\ n \mid x + 1 \mid \epsilon\rangle, \langle x \mapsto fact\ n \mid x + 2 \mid \epsilon\rangle)$$

Unfortunately, this choice leads to duplication of the expensive $fact\ n$ subterm. If we freely duplicate unbounded amounts of work in this manner we can easily end up "optimising" the program into a much less efficient version.

Work can be duplicated even if no syntactic duplication occurs, as occurs if we take this example:

$$\langle x \mapsto fact\ n \mid \lambda y.\, x + y \mid \epsilon\rangle$$

And split it as follows:

$$\lambda y \to \langle x \mapsto fact\ n \mid x + y \mid \epsilon\rangle$$

Furthermore, syntactic duplication does not necessarily lead to work duplication. Consider:

$$\langle x \mapsto fact\ n \mid y \mid \textbf{case} \bullet \textbf{of}\ (True \to x + 1; False \to x + 2)\rangle$$

Notice that splitting it as follows does not duplicate the computation of $fact\ n$:

$$\begin{aligned}&\textbf{case}\ y\ \textbf{of}\ True \ \to \langle x \mapsto fact\ n \mid x + 1 \mid \epsilon\rangle\\ &\qquad\qquad\ False \to \langle x \mapsto fact\ n \mid x + 2 \mid \epsilon\rangle\end{aligned}$$

Consequently, we push the heap bindings supplied to $split$ down into those split-out subterms of which they are free variables, as long as either one of these conditions is met:

- The binding manifestly binds a value, such as $\lambda x.\, x$: values require no further reduction, so no work can be lost that way
- Pushing the binding down into the subterm would not result in the allocation of its thunk occurring more than once in any possible context consuming the output

Our $split$ uses **let**-floating to make more heap bindings suitable for pushing down under these criteria. For example, this state:

$$\langle x \mapsto Just\ (fact\ n) \mid \lambda m.\, \textbf{case}\ x\ \textbf{of}\ Just\ y \to y + m \mid \epsilon\rangle$$

Will be split as follows:

$$\begin{aligned}&\textbf{let}\ a = \langle \epsilon \mid fact\ n \mid \epsilon\rangle\\ &\textbf{in} \lambda m.\, \langle x \mapsto Just\ a \mid \textbf{case}\ x\ \textbf{of}\ Just\ y \to y + m \mid \epsilon\rangle\end{aligned}$$

Sketching $split$ Due to space limitations, we are unable to give a complete description of $split$. However, we can give a sketch of a suboptimal implementation that may nonetheless clarify our description.

We first introduce the concept of a $Bracket$. This is a Haskell representation of the "term with holes" notational device we introduced earlier. Each hole contains a $State$:

$$\begin{aligned}&\textbf{data}\ Bracket = B\ \{holes :: [State],\\ &\qquad\qquad\qquad\qquad assemble :: [Term] \to Term\}\\ &termBracket :: Term \to Bracket\\ &termBracket\ e = B\ [(emptyHeap, e, emptyStack)]\ (\lambda[e'] \to e')\end{aligned}$$

Our code examples will often make use of a $[\![$bracketed$]\!]$ syntax to concisely define a value of type $Bracket$:

$$[\![f\ \ \langle \epsilon \mid 1 \mid \epsilon\rangle]\!] :: Bracket$$

This particular example corresponds to:

$$B\ \{holes = [(\epsilon, 1, \epsilon)], assemble = \lambda[e'] \to var\ \texttt{"f"}\ \text{`}apps\text{`}\ e'\}$$

Split can now be defined as follows:

$$\begin{aligned}&split\ opt\ (h, e, k) = liftM\ (assemble\ br)\ \$\ mapM\ opt\ (holes\ br)\\ &\quad \textbf{where}\\ &\quad\quad xs = \textbf{case}\ e\ \textbf{of}\ \ Var\ x \to [x];\ \_ \to [\,]\\ &\quad\quad br = splitHeap\ h\ \$\ splitStack\ xs\ k\ \$\ splitTerm\ e\end{aligned}$$

Each part of the $State$ is split independently to produce a $Bracket$, which than has all of it's $holes$ optimised before we rebuild the final term. Before we cover $splitTerm$, $splitStack$ and $splitHeap$, we will need a way to build a larger bracket from smaller ones:

$$\begin{aligned}&plusBrackets :: [Bracket] \to ([Term] \to Term) \to Bracket\\ &plusBrackets\ brs\ rb = B\ \{holes = concatMap\ holes\ brs,\\ &\qquad\qquad\qquad\qquad\qquad\qquad assemble = f\}\\ &\quad \textbf{where}\\ &\quad\quad f\ es = rb\ (zipWith\ (\lambda br\ es \to assemble\ br\ es)\ brs\ ess)\\ &\quad\quad\quad \textbf{where}\ ess = splitManyBy\ (map\ holes\ brs)\ es\\ &splitManyBy :: [[b]] \to [a] \to [[a]]\\ &\quad \text{-- } splitManyBy\ bss\ as \equiv ass \wedge length\ (concat\ bss) \equiv length\ as\\ &\quad \text{-- } \implies map\ length\ bss \equiv map\ length\ ass \wedge as \equiv concat\ ass\end{aligned}$$

Now, $splitTerm$ just identifies some subexpressions for supercompilation:

$$\begin{aligned}&splitTerm :: Term \to Bracket\\ &splitTerm\ e = plusBrackets\ (map\ termBracket\ es)\ rb\\ &\quad \textbf{where}\ (es, rb) = uniplate\ e\end{aligned}$$

We make use of the $uniplate$ combinator (following Mitchell and Runciman [8]), which takes a $Term$ apart into a list of its immediate subterms, and a function to recombine those subterms to obtain the original input:

$$uniplate :: Term \to ([Term], [Term] \to Term)$$

There is more work to do when splitting the stack:

$$\begin{aligned}&splitStack :: [Var] \to Stack\\ &\qquad\qquad\quad \to Bracket\\ &\qquad\qquad\quad \to ([(Var, Bracket)], Bracket)\end{aligned}$$

The call $splitStack\ \ xs\ \ k\ \ b$ splits stack k with bracket b in the focus, where all of the variables xs are guaranteed to have the same value as the focus. We will use the xs in $splitStack$ to learn from residual **case** branches.

There are three principal possibilities that $splitStack$ has to deal with. Firstly, applications and primitives can be handled uniformly:

$$\begin{aligned}&splitStack\ xs\ (\bullet\ x : k)\ br = splitStack\ [\,]\ k\ \ [\![\langle br\rangle\ x]\!]\\ &splitStack\ xs\ (\bullet \otimes\ e : k)\ br\\ &\quad = splitStack\ [\,]\ k\ \ [\![\langle br\rangle \otimes\ \langle \epsilon \mid e \mid \epsilon\rangle]\!]\\ &splitStack\ xs\ (v \otimes \bullet : k)\ br\\ &\quad = splitStack\ [\,]\ k\ \ [\![\langle \epsilon \mid v \mid \epsilon\rangle \otimes\ \langle br\rangle]\!]\end{aligned}$$

The next possibility is that the stack frame arises from a **case**:

$$\begin{aligned}&splitStack\ xs\ (\textbf{case} \bullet \textbf{of} \overline{\alpha \to e} : k)\ br\\ &\quad = ([\,], \Big[\!\!\Big[\ \textbf{case}\ \langle br\rangle\ \textbf{of}\ \overline{\alpha \to \langle altbr\rangle}\Big]\!\!\Big])\\ &\quad \textbf{where}\\ &\quad\quad \overline{altbr} = \overline{\langle altHeap\ \alpha \mid e \mid k\rangle}\\ &\quad\quad altHeap\ \alpha = fromList\ [(x, altConValue\ \alpha) \mid x \leftarrow xs]\\ &\quad\quad altConValue :: AltCon \to Value\\ &\quad\quad altConValue\ (\mathbf{C}\ \overline{x}) = (\mathbf{C}\ \overline{x})\\ &\quad\quad altConValue\ \ell \qquad = \ell\end{aligned}$$

Notice that we *do not* recursively call $splitStack$ in this situation: as we discussed, the entire stack is pushed into each case branch. We also use $altHeap$ to construct a heap that binds the variables being scrutinised (if any) to the value corresponding to the particular case alternative.

Finally, the immediate stack frame may be an update frame:

$$
\begin{aligned}
&splitStack\ xs\ (\mathbf{update}\ x : k)\ br\\
&\quad = ((x, br) : xbrs', br')\\
&\quad \mathbf{where}\ (xbrs', br') = splitStack\ (x : xs)\ k\ [\![x]\!]
\end{aligned}
$$

In this case, we recursively split the remainder of the stack, but change the focus to be the *variable being updated*. The presence of update frames is why $splitStack$ returns a $[(Var, Bracket)]$ as well as a $Bracket$ – the list of $(Var, Bracket)$ contains a $Bracket$ for every update frame that $splitStack$ encountered. As we will see shortly, the brackets from this list will be placed in an enclosing **let** expression along with those arising from the $Heap$.

Finally, we can implement $splitHeap$:

$$
\begin{aligned}
&splitHeap :: Heap\\
&\qquad\qquad \to ([(Var, Bracket)], Bracket)\\
&\qquad\qquad \to Bracket\\
&splitHeap\ h\ (xbrs, br)\\
&\quad = plusBrackets\ (map\ inline\ (br : brs))\\
&\qquad\qquad\qquad (\lambda(e : es) \to letRec\ (xs\ \text{`}zip\text{'}\ es)\ e)\\
&\quad \mathbf{where}\ (xs, brs) = unzip\ (xbrs +\!\!+\ [\ (x, termBracket\ e)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid (x, e) \leftarrow toList\ h])
\end{aligned}
$$

This completes the implementation of $split$. A real implementation will need to add several complications:

- The $splitHeap$ function should attempt to push some elements of the $Heap$ into the $holes$ of the brackets from $splitStack$. A linearity analysis will be required in order to avoid duplicating work when non-value heap bindings get pushed down.
- The $Heap$ should be let-floated to expose values under **let**s, and hence allow more bindings to be propagated downwards.
- In the presence of recursive **let** it is not always valid for $splitStack$ to push down the entire stack into the branches of a residual **case**. This issue is discussed in more detail in Section 4.

3.6 Termination of the supercompiler

Although we have been careful to ensure that our evaluation function, $reduce$, is total, it is not so obvious that sc itself is terminating. Since $split$ may recursively invoke sc via its higher order argument, we might get an infinitely deep stack of calls to sc!

To rule out this possibility, sc carries a history, which – as we saw in Section 3 – is checked before any reduction is performed. If $terminate$ allows the history to be extended, the input $State$ is reduced before recursing. Otherwise, the input $State$ is fed to $split$ unchanged.

In order to be able to prove that the supercompiler terminates, we need some condition on exactly what sort of subcomponents $split\ opt$ invokes opt on. It turns out that the presence of recursive let requires us to choose a rather complicated condition here, as we will explain further in Section 4.4.

Let us pretend for a moment that we have no recursive **let**. In this scenario, it is always the case for our $split$ that $split\ \ opt\ \ s$ invokes $opt\ \ s'$ only if $s' \prec s$. The $\prec$ relation is a well-founded relation defined by $s' \prec s \iff size(s') < size(s)$, where $size : State \to \mathbb{N}$ returns the number of abstract syntax tree nodes in the $State$. This is sufficient to ensure termination, as the following argument shows:

Theorem: sc always recurses a finite number of times Proceed by contradiction. If sc recursed an infinite number of times, then by definition the call stack would contain infinitely many activations of $sc\ \ hist\ \ s$ for (possibly repeating) sequences of $hist$ and s values. Denote the infinite chains formed by those values as $\langle hist_0, hist_1, \ldots\rangle$ and $\langle s_0, s_1, \ldots\rangle$ respectively.

Now, observe that there must be infinitely many i such that $isStep\ \ (terminate\ hist_i\ s_i)$. This follows because the only other possibility is that there must exist some j such that $\forall l. l \geq j \implies isStep\ \ (terminate\ hist_l\ s_l)$. On such a suffix, sc is recursing through $split$ without any intervening uses of $reduce$. However, by the property we required $split$ to have, such a sequence of states must have a strictly decreasing size:

$$\forall l. l > j \implies size\,(s_l) < size\,(s_j)$$

However, $<$ is a well founded relation, so such a chain cannot be infinite. This contradicts our assumption that this suffix of sc calls is infinite, so it must be the case that there are infinitely many i such that $isContinue\ \ (terminate\ hist_i\ s_i)$.

Now, form the infinite chain $\langle t_1, t_2, \ldots\rangle$ consisting of s_i such that $isContinue\ \ (terminate\ hist_i\ s_i)$. By the properties of $terminate$, it follows that $\forall ij. j < i \implies \neg\,(tagBag\ t_j \lhd tagBag\ t_i)$. However, this contradicts the fact that $\lhd$ is a well-quasi-order.

$\square$

Combined with the requirement that $split\ \ opt$ only calls opt finitely many times, the whole supercompilation process must terminate.

Two non-termination checks It is important to note that the history carried by sc is extended entirely independently from the history produced by the $reduce$ function. The two histories deal with different sources of non-termination.

The history carried by $reduce$ prevents non-termination due to divergent expressions, such as this one:

$$\mathbf{let}\ f\ x = 1 + (f\ x)\ \mathbf{in}\ f\ 10$$

In contrast, the history carried by sc prevents non-termination that can arise from repeatedly invoking the $split$ function – even if every subexpression would, considered in isolation, terminate. This is illustrated in the following program:

$$\mathbf{let}\ count\ n = n : count\ (n+1)\ \mathbf{in}\ count\ 0$$

Left unchecked, we would repeatedly $reduce$ the calls to $count$, yielding a value (a cons-cell) each time. The $split$ function would then pick out both the head and tail of the cons cell to be recursively supercompiled, leading to yet another unfolding of $count$, and so on. The resulting (infinite) residual program would look something like:

$$
\begin{aligned}
\mathbf{let}\ & h0 = h1 : h2; h1 = 0\\
& h2 = h3 : h4; h3 = 1\\
& h4 = h5 : h6; h5 = 2\\
& \ldots
\end{aligned}
$$

The check with $terminate$ before reduction ensures that instead, one of the applications of $count$ is left unreduced. This use of $terminate$ ensures that our program remains finite:

$$
\begin{aligned}
\mathbf{let}\ & h0 = h1 : h2; h1 = 0\\
& h2 = \mathbf{let}\ count = \lambda n.\, h3\ n\\
& \qquad\ \mathbf{in}\ count\ 1\\
& h3\ n = n : h3\ (n+1)\\
\mathbf{in}\ & h0
\end{aligned}
$$

Negative recursion in data constructors As a nice aside, the rigorous termination criteria gives us a stronger termination guarantee than the Glasgow Haskell Compiler (GHC) [9], the leading

Haskell implementation. Because GHC does not check for recursion through negative positions in data constructors, the following notorious program will force GHC into an infinite loop:

$$\begin{aligned}&\mathbf{data}\ U = \mathit{MkU}\ (U \to \mathit{Bool})\\ &\mathit{russel}\ u@(\mathit{MkU}\ p) = \mathit{not}\ (p\ u)\\ &x = \mathit{russel}\ (\mathit{MkU}\ \mathit{russel}) :: \mathit{Bool}\end{aligned}$$

3.7 Observations on the basic supercompiler

It is a unique feature of our supercompiler that all our ingredients operate on States, rather than Terms. This is a consequence of explicitly basing the supercompiler on an evaluator, but it pays off in two other ways as well:

1. The memoiser (Section 3.4) matches States rather than Terms. This is beneficial because States can be thought of as Terms that have been weakly normalised by evaluation – two States with equal semantics are more likely to match than two Terms with equal semantics.
2. The splitter (Section 3.5) operates distinctively differently on each of the three components of the State. To split a Term well would be much harder.

4. Extending to recursive let

In the previous section, we described all the pieces necessary to implement a complete supercompiler. The handling of recursive **let** is mostly straightforward in this framework, with the exception of two things:

- Update frames originating from recursive **let** complicate the splitter: Section 4.3
- The termination proof for the supercompiler becomes more complicated: Section 4.4

We cover each of these points in order.

4.1 Update frames

The evaluator (Figure 3 and Section 3.3) deals with a call-by-need language, using *update frames* in the conventional way to model laziness [6]. When a heap binding $x \mapsto e$ is demanded by a variable x coming into the focus of the evaluator, e may not yet be a value. To ensure that we only reduce any given heap-bound e to a value at most once, the evaluator pushes an update frame **update** x on the stack, before beginning the evaluation of e . After e has been reduced to a value, v, the update frame will be popped from the stack, which is the cue for the evaluator to update the heap with a binding $x \mapsto v$, replacing the old one. Now, subsequent uses of x in the course of evaluation will be able to reuse that value directly, without reducing e again.

As an example of how update frames work, consider this reduction sequence:

$$\begin{aligned}&\langle x \mapsto 1+2 \mid x+x \mid \epsilon\rangle \rightsquigarrow \langle x \mapsto 1+2 \mid x \mid \bullet + x\rangle\\ &\quad\rightsquigarrow \langle \epsilon \mid 1+2 \mid \mathbf{update}\ x, \bullet + x\rangle \rightsquigarrow \ldots\\ &\quad\rightsquigarrow \langle \epsilon \mid 3 \mid \mathbf{update}\ x, \bullet + x\rangle \rightsquigarrow \langle x \mapsto 3 \mid 3 \mid \bullet + x\rangle\\ &\quad\rightsquigarrow \langle x \mapsto 3 \mid x \mid 3 + \bullet\rangle \rightsquigarrow \langle \epsilon \mid 3 \mid \mathbf{update}\ x, 3 + \bullet\rangle\\ &\quad\rightsquigarrow \langle x \mapsto 3 \mid 3 \mid 3 + \bullet\rangle \rightsquigarrow \langle x \mapsto 3 \mid 6 \mid \epsilon\rangle\end{aligned}$$

Because the corresponding heap binding is removed from the heap whenever an update frame is pushed, the update frame mechanism is what causes reduction to become blocked if you evaluate a term which forms a black hole:

$$\langle x \mapsto x+1 \mid x \mid \epsilon\rangle \rightsquigarrow \ldots \rightsquigarrow \langle \epsilon \mid x \mid \bullet + 1, \mathbf{update}\ x\rangle \not\rightsquigarrow$$

Update frames complicate the supercompiler slightly, but in a localised way – we must think carefully as to how the *split* function should deal with update frames.

4.2 Splitting in the presence of update frames

Just like all other kinds of stack frame, we want to push update frames into residual case branches. Consider this input to *split*:

$$\langle \epsilon \mid x \mid \mathbf{case}\bullet\mathbf{of}\ T \to F, \mathbf{update}\ y, \mathbf{case}\bullet\mathbf{of}\ F \to (2,y)\rangle$$

We will *split* as follows, pushing the whole stack, including the update frame for y, into the **case** branch:

$$\mathbf{case}\ x\ \mathbf{of}\ T \to \langle \epsilon \mid F \mid \mathbf{update}\ y, \mathbf{case}\ \bullet\ \mathbf{of}\ F \to (2,y)\rangle$$

After supercompilation is complete, we will then obtain an output term something like the following:

$$\mathbf{case}\ x\ \mathbf{of}\ T \to \mathbf{let}\ y = F\ \mathbf{in}\ (2,y)$$

This is what the *splitStack* function we saw in Section 3.5 does.

4.3 Splitting update frames from recursive lets

The key problem that the splitter must face is that update frames derived from recursive **let** can interact badly with our intention to push the entire enclosing stack into the branches of a **case**. Consider this input to *split*:

$$\langle \epsilon \mid \mathit{unk} \mid \bullet + y, \mathbf{case}\ \bullet\ \mathbf{of}\ 1 \to 2, \mathbf{update}\ y, \bullet + 2\rangle$$

Following our earlier discussion of **case**, we might be tempted to split as follows:

$$\mathbf{case}\ \mathit{unk} + y\ \mathbf{of}\ 1 \to \langle \epsilon \mid 2 \mid \mathbf{update}\ y, \bullet + 2\rangle$$

However, this is a disastrous choice – due to the occurrence of y in the scrutinee, y is now a free variable of the output expression! The lesson here is that update frames should not be pushed inside case branches if they bind a variable that we may need to refer to outside the **case**. Following this rule, our example is instead *split* as follows:

$$\begin{aligned}&\mathbf{let}\ y = \mathbf{case}\ \mathit{unk} + y\ \mathbf{of}\ 1 \to \langle \epsilon \mid 2 \mid \epsilon\rangle\\ &\mathbf{in}\ y + \langle \epsilon \mid 2 \mid \epsilon\rangle\end{aligned}$$

Irritatingly, the choice about which update frames should not be pushed inside **case** branches is not as straightforward as a simple free-variable check. The reason is that choosing to not push an update frame down may make more of the variables bound by other pushable update frames free, and hence require us to prevent pushing in yet more update frames! Here is a contrived example illustrating the point – note that for clarity we will not write the update frames directly, and represent the States as if they were terms:

$$\begin{aligned}&\mathbf{let}\ w = \mathit{fact}\ z; y = \mathit{unk} + x\\ &\quad\ \ x = \mathbf{case}\ y\ \mathbf{of}\ 10 \to w + 3\\ &\quad\ \ z = \mathbf{case}\ x\ \mathbf{of}\ 20 \to a + 3\\ &\mathbf{in}\ z + w + a\end{aligned}$$

Our initial guess at the output of *split* may be as follows:

$$\begin{aligned}&\mathbf{let}\ y = \mathit{unk} + \langle x\rangle\\ &\mathbf{in}\ \mathbf{case}\ y\ \mathbf{of}\\ &\quad 10 \to \langle\ \mathbf{let}\ w = \mathit{fact}\ z; x = w + 3\\ &\qquad\qquad\quad z = \mathbf{case}\ x\ \mathbf{of}\ 20 \to a + 3\\ &\qquad\quad\ \ \mathbf{in}\ z + w + a\ \rangle\end{aligned}$$

Unfortunately, x is now a free variable of the whole expression, and consequently we should not have pushed the update frame for x within the **case** branch. Based on this information, our next guess may be:

$$\begin{aligned}&\mathbf{let}\ w = \langle \mathit{fact}\ z\rangle; y = \mathit{unk} + \langle x\rangle\\ &\quad\ \ x = \mathbf{case}\ y\ \mathbf{of}\ 10 \to \langle w + 3\rangle\\ &\mathbf{in}\ \mathbf{case}\ x\ \mathbf{of}\ 20 \to \langle\ \mathbf{let}\ z = a + 3\\ &\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ z + w + a\ \rangle\end{aligned}$$

Note that we have now been forced not to push the w binding down into either the **case** branch, because doing so would risk work duplication. Unfortunately, that has caused z to be free in the output expression! The correct solution is in fact to not push down the update frames for *both* x and z:

$$\begin{array}{l}\textbf{let}\ w = \langle \mathit{fact}\ z\rangle; y = \mathit{unk} + \langle x\rangle \\ \quad\ x = \textbf{case}\ y\ \textbf{of}\ 10 \to \langle w+3\rangle \\ \quad\ z = \textbf{case}\ x\ \textbf{of}\ 20 \to \langle a+3\rangle \\ \textbf{in}\ z + \langle w\rangle + \langle a\rangle\end{array}$$

Our real *split* implementation uses a fixed point that follows essentially this reasoning process to determine the set of update frames which may not be pushed down.

4.4 Termination in the presence of recursive let

In Section 3.6 we showed why the supercompiler without recursive **let** terminated. However, to make that argument we had to rely on a condition on *split* that is simply too restrictive for the supercompiler with recursive **let**.

Before, we used the property that $\mathit{split}\ \mathit{opt}\ s$ invoked $\mathit{opt}\ s'$ only if $s' \prec s \iff \mathit{size}\,(s') < \mathit{size}\,(s)$. However, consider this input to *split*:

$$\langle f \mapsto \lambda y.\, \mathit{Just}\ (f\ (\mathit{not}\ y)) \mid \mathit{Just}\ (f\ (\mathit{not}\ y)) \mid \epsilon\rangle$$

We would like to *split* as follows:

$$\begin{array}{l}\textbf{let}\ f = \lambda x.\ \langle f \mapsto \lambda y.\, \mathit{Just}\ (f\ (\mathit{not}\ y)) \mid \mathit{Just}\ (f\ (\mathit{not}\ y)) \mid \epsilon\rangle \\ \textbf{in}\ \mathit{Just}\ (f\ (\mathit{not}\ y))\end{array}$$

This is disallowed by the *size*-based criteria because the recursively-optimised State would be no smaller than the input.

In the presence of recursive **let**, we can instead use the property that for our *split*, $\mathit{split}\ \mathit{opt}\ (h, e, k)$ only invokes opt on states (h', e', k') that satisfy all of these conditions:

1. $h' \subseteq h \cup \textit{alt-heap}\,(e, k)$
2. k' ‘$\mathit{isInfixOf}$‘ k
3. $e' \in \mathit{subterms}\,(h, e, k)$

The $\mathit{subterms}\,(h, e, k)$ function returns all expressions that occur syntactically within any of the Heap, Stack or Term inputs. The $\textit{alt-heap}\,(e, k)$ function takes the variables bound by update frames in k and, if $e \equiv \mathit{Var}\ x$, the variable x. It then forms the cross product of that set with the values corresponding to the α in any $\textbf{case} \bullet \textbf{of} \overline{\alpha \to e} \in k$.

We are now in a position to repair the proof.

Theorem: sc ***always recurses a finite number of times*** Proceed by contradiction. If sc recursed an infinite number of times, then by definition the call stack would contain infinitely many activations of $sc\ \mathit{hist}\ s$ for (possibly repeating) sequences of hist and s values. Denote the infinite chains formed by those values as $\langle \mathit{hist}_0, \mathit{hist}_1, \ldots\rangle$ and $\langle s_0, s_1, \ldots\rangle$ respectively.

Now, observe that there must be infinitely many i such that $\mathit{isStep}\ \ (\mathit{terminate}\ \mathit{hist}_i\ s_i)$. This follows because the only other possibility is that there must exist some j such that $\forall l. l \geq j \implies \mathit{isStep}\ \ (\mathit{terminate}\ \mathit{hist}_l\ s_l)$. On such a suffix, sc is recursing through *split* without any intervening uses of *reduce*. By the modified property of *split* and the properties of *alt-heap* and *subterms* we have that

$$\begin{array}{ll}\forall l. l \geq j \implies & \\ & h_l \subseteq h_j \cup \textit{alt-heap}\,(e_j, k_j) \\ \wedge & k_l\ \text{‘}\mathit{isInfixOf}\text{‘}\ k_j \\ \wedge & e_l \in \mathit{subterms}\,(s_j)\end{array}$$

We can therefore conclude that the infinite suffix must repeat itself at some point: $\exists l. l > j \wedge s_l \equiv s_j$. However, we required that match always succeeds when matching two terms equivalent up to renaming, which means that $sc\ \mathit{hist}_l\ s_l$ would have been tied back by memo rather than recursing. This contradicts our assumption that this suffix of sc calls is infinite, so it must be the case that there are infinitely many i such that $\mathit{isContinue}\ \ (\mathit{terminate}\ \mathit{hist}_i\ s_i)$.

Now, form the infinite chain $\langle t_1, t_2, \ldots\rangle$ consisting of s_i such that $\mathit{isContinue}\ \ (\mathit{terminate}\ \mathit{hist}_i\ s_i)$. By the properties of $\mathit{terminate}$, it follows that $\forall ij. j < i \implies \neg\,(\mathit{tagBag}\ t_j \lhd \mathit{tagBag}\ t_i)$. However, this contradicts the fact that $\lhd$ is a well-quasi-order.

□

Although the termination argument becomes more complex, the actual supercompilation algorithm remains as simple and beautiful as ever.

5. Results

We have implemented the supercompiler for a subset of Haskell. it is implemented as a preprocessor: programs are run through the supercompiler before being compiled by GHC at the -O2 optimisation level. The preliminary results of running the supercompiler on a standard array of benchmark programs are shown in Figure 4. For comparison, we include benchmark results from a supercompiler of Mitchell [4].

The “append”, “factorial”, “raytracer”, “sumtree” and “treeflip” benchmarks are all standard examples that have been described in previous work on supercompilation and deforestation [4, 10, 2, 11]. The “sumsquare” program is taken from work in stream fusion [12]. The “bernouilli”, “digitsofe2”, “exp3_8”, “primes”, “rfib”, “tak”, “wheel-sieve1”, “wheel-sieve2” and “x2n1” benchmarks are from the imaginary portion of the nofib benchmark suite [13].

We tested two variants of our supercompiler: one where we the supercompiler evaluated primitive operations (primops), and one where it did not. Both variants treated primitives as strict operations.

The benchmark results are promising. The supercompiler without primops reduced runtime by an average[3] of 70% compared to GHC alone. Evaluating primops reduced average speedup to 57%. Mitchell’s system achieved an average speedup of 51%.

The use of supercompilation in practice is limited because despite the fact that it is a guaranteed to terminate, it might take very long indeed to do so. Nofib imaginary suite benchmarks such as “digitsofe1” and “gen_regexps” are prohibitively expensive to supercompile in both our system and that of Mitchell. Interestingly, the same problem afflicts “tak” – but only when evaluation of primops is enabled.

Primitive operations Indeed, the supercompiler performed worse overall when evaluating primops than when it left them unevaluated – particularly suffering on “sumtree” and “treeflip”. These benchmarks have a common structure where a binary tree is generated and then consumed by a function pipeline, terminate by a simple sum of the tree nodes. The initial construction of the tree does not deforest cleanly, but the consuming function pipeline makes several intermediate copies of the tree which can be deforested to produce a function that produces the required sum directly. Both our system (without primops) and Mitchell’s system are able to fuse these pipelines together.

The addition of primops to the system means that we create specialisations of the fused pipeline that include in their evaluation contexts frames such as $2 + \bullet$, where 2 is a partial sum of the tree. Every specialisation of the fused pipeline includes such a stack frame, and because the partial sum changes regularly those specialisations can never be reused. We end up building a lot of specialisations of the pipeline for a few values of the partial sum,

[3] Averages are computed using a harmonic mean

Program	Mitchell [4]					Evaluator-based, no primops					Evaluator-based, primops				
	SC.[a]	Cmp.[b]	Run[c]	Mem[d]	Size[e]	SC.[a]	Cmp.[b]	Run[c]	Mem[d]	Size[e]	SC.[a]	Cmp.[b]	Run[c]	Mem[d]	Size[e]
append	0.0s	0.88	0.86	0.85	1.29	0.0s	1.00	0.89	0.87	3.24	0.0s	1.03	0.92	0.87	3.24
bernouilli	5.8s	1.63	0.98	0.97	3.76	0.1s	1.07	0.98	0.95	2.26	0.1s	1.07	0.98	0.95	2.24
digitsofe2	4.2s	1.24	0.32	0.46	1.15	0.1s	1.07	1.17	1.08	2.81	0.1s	1.08	1.18	1.09	2.79
exp3_8	0.8s	1.34	0.96	1.00	6.59	8.7s	2.85	0.59	0.67	85.17	15.4s	3.35	0.55	0.67	114.31
factorial	0.0s	0.99	0.95	1.00	0.77	0.0s	0.96	0.99	1.00	1.00	0.0s	0.98	1.05	1.00	0.91
primes	0.1s	1.04	0.63	0.99	0.79	0.0s	0.98	0.72	1.07	0.87	0.0s	0.98	0.71	1.07	0.80
raytracer	0.0s	1.00	0.57	0.44	1.54	0.0s	1.00	0.52	0.45	1.37	0.0s	1.00	0.51	0.45	1.38
rfib	0.0s	0.94	0.93	1.00	0.87	0.0s	1.00	0.67	1.00	2.00	0.0s	1.00	0.67	1.01	2.00
sumsquare	19.5s	1.45	0.36	0.00	7.38	2.3s	1.97	0.05	0.00	20.78	3.0s	1.95	0.06	0.00	21.15
sumtree	0.1s	1.01	0.13	0.00	1.50	0.0s	1.02	0.14	0.00	2.46	0.2s	1.24	0.68	0.93	9.09
tak	0.1s	0.86	0.81	655.04	0.59	0.1s	1.34	0.74	18644.34	7.22	N/A	N/A	N/A	N/A	N/A
treeflip	0.1s	1.03	0.56	0.45	1.99	0.0s	1.02	0.13	0.05	2.53	0.2s	1.47	0.81	0.91	19.40
wheel-sieve1	N/A	N/A	N/A	N/A	N/A	22.2s	7.87	0.90	0.53	71.07	16.8s	10.61	1.00	0.54	71.47
wheel-sieve2	N/A	N/A	N/A	N/A	N/A	1.3s	3.16	1.55	1.21	18.35	1.4s	3.06	1.55	1.21	18.24
x2n1	0.1s	1.06	0.92	0.99	1.39	0.0s	1.10	0.99	0.95	1.21	0.0s	1.15	0.99	0.95	1.18

[a] Supercompilation time (seconds) [b] GHC runtime relative to no supercompilation [c] Program runtime relative to no supercompilation
[d] Allocation relative to no supercompilation [e] Size (in syntax tree nodes) of program relative to no supercompilation

Figure 4: Benchmark results

before the termination condition kicks in and stops us. Unfortunately, the resulting termination splitting prevents us from fusing the pipeline *entirely*. The net result is that the first few iterations of the sum are computed with perfect deforestation, but later iterations must fall back on a fully-forested function isomorphic to the original unfused pipeline.

***Recursive* let** We are able to report results for two benchmarks ("wheel-sieve1" and "wheel-sieve2") that Mitchell's system is unable to supercompile because they make fundamental use of recursive let. We achieve an improvement in "wheel-sieve1" by deforesting intermediate lists, but actually manage to *increase* allocations in "wheel-sieve2".

Opportunities for improvement The "tak" benchmark reported a staggering 18,000-fold increase in allocations, although this was up from a very low base – the unmodified program allocates only 13kB. Mitchell's supercompiler exhibits the same problem, albeit to a lesser degree. Investigation shows that the allocation increase is due to supercompilation introducing several large *join points* which take boxed integers as arguments. When compiled without supercompilation, there are no join points and all arithmetic is unboxed by GHC's strictness analyser [14].

The benchmark where we do noticeably worse than Mitchell is "digitsofe2" – we actually increase both allocations and runtime, while he reduces each figure by more than 50%. Although the exact reasons remain unclear, it appears that once again the problem is that the supercompilation process has prevented GHC from aggressively unboxing the output.

Supercompilation time Benchmarking our supercompiler on one program ("digits-of-e2") showed that the vast majority of time (42%) is spent on managing names and renaming. Matching against previous states accounted for 14% of the runtime. Only 6% of time was spent testing the termination condition.

6. Related Work

Supercompilation was introduced by Turchin [1], but has recently seen a revival of interest from both the call-by-value [15, 10] and call-by-need [4] perspectives.

Partial evaluation [16] is a technique closely related to supercompilation. The fields overlap somewhat, but supercompilers tend to make a distinctive set of choices which set them apart: they specialise expressions in the context in which they occur, operate on unannotated programs and test for termination online. Theoretical work has suggested that certain kinds of partial evaluator suffer from strictly less information propagation than supercompilers, limiting their optimising power [17].

The idea of building a partial evaluation system around an actual evaluator is hardly new – it is present from the very earliest work by Sestoft et al. [18]. However, this approach seems to have received surprisingly little attention in the supercompilation community.

Much of the supercompilation literature makes use of the *homeomorphic embedding* test for ensuring termination [10, 19, 15]. Users of this test uniformly report that testing the termination condition makes up the majority of their supercompilers runtime [10, 19]. The tag-bag criteria appears to be much more efficient in practice, as our supercompiler spends only 6% of its runtime testing the criteria.

Jørgensen has previously produced a compiler for call-by-need through partial evaluation of a Scheme partial evaluator with respect to an interpreter for the lazy language [20]. His work made use of a partial evaluator capable of dealing with the $set!$ primitive, which was used to implement updateable thunks. Our supercompiler avoids the need for any imperative features in the language being supercompiled, and deals with the call-by-need evaluation order directly.

7. Further Work

The major barriers to the use of supercompilation in practice are code bloat and compilation time. One method to achieve an improvement in both dimensions would be to reuse specialisations more aggressively. For example, consider the following program:

$$\begin{array}{l}\mathbf{let}\ \mathit{replicate} = \lambda n\ y.\ \mathbf{if}\ n \leqslant 0\ \mathbf{then}\ [\,] \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ y : \mathit{replicate}\ (n-1)\ y \\ \mathbf{in}\ (\mathit{replicate}\ 4\ \texttt{'c'}, \mathit{replicate}\ 4\ \texttt{'d'})\end{array}$$

During the reduction of $\mathit{replicate}\ n\ y$, the evaluator never needs to use the definition of y in order to achieve reduction. Nonetheless, existing supercompilers – including the one described here – will (modulo termination checking) duplicate the whole $\mathit{replicate}$ call for no real gain:

$$\begin{array}{l}\mathbf{let}\ \mathit{h0} = (\mathit{h1}, \mathit{h2}) \\ \quad\ \mathit{h1} = \texttt{'c'} : \texttt{'c'} : \texttt{'c'} : \texttt{'c'} : [\,] \\ \quad\ \mathit{h2} = \texttt{'d'} : \texttt{'d'} : \texttt{'d'} : \texttt{'d'} : [\,] \\ \mathbf{in}\ \mathit{h0}\end{array}$$

However, we do not sacrifice any optimisation opportunities if were to instead produce the following output program:

$$
\begin{aligned}
&\mathbf{let}\ h0 = (h1\ \texttt{'c'}, h1\ \texttt{'d'})\\
&\quad\ \ h1 = \lambda y.\, y : y : y : y : [\,]\\
&\mathbf{in}\ h0
\end{aligned}
$$

We have a partial implementation of a system that achieves this additional code sharing and prevents over-allocation through a unified mechanism, and intend to report on our experience with it shortly.

Because the supercompiler described here is nicely separated from issues of evaluation order, it should be straightforward to modify the system to supercompile a pure call-by-value language such as Timber [21]. The only substantial work required would be modifying in *split* to deal with the kinds of evaluation context arising from call-by-value reduction. However, a splitter for call-by-value (or call-by-name) is rather simple to define because such evaluation strategies have no equivalent to update frames, and it is always permissible to duplicate heap bindings – so no work-duplication check is required at all. We speculate that an adaption of our supercompiler to call-by-value would yield a supercompiler with similar power to recently reported results of Jonsson and Nordlander [22].

We plan to extend the supercompiler to work on the typed language System FC [23] for implementation as a part of GHC. Again, this should be fairly straightforward, and involve mostly local changes to the evaluator. Supercompilation works best when it has access to the whole program, but GHC already has the necessary facilities to get hold of the definitions from imported modules, in the shape of interface files.

8. Conclusions

Supercompilation is a simple, powerful and principled technique for program optimisation. A single pass with a supercompiler achieves many optimisations that have traditionally been laboriously specified and implemented independently.

We have shown how to produce a supercompiler by basing it explicitly on an evaluator. This clean design allowed us to extend the technique to lazy languages with recursive **let**, by building the supercompiler around a call-by-need evaluator.

Initial benchmark results are promising, but also bring to light weaknesses in the algorithm. In particular, a method is sorely needed for reducing the worst-case runtime of supercompilation.

Acknowledgments

This work was partly supported by a PhD studentship generously provided by Microsoft Research. Thanks are due to Neil Mitchell and Peter Jonsson for enlightening discussions and feedback.

References

[1] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[2] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Berlin / Heidelberg, 1988.

[3] Simon Peyton Jones. Constructor specialisation for Haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 327–337, 2007.

[4] Neil Mitchell. Rethinking supercompilation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*. ACM, 2010.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 28(6):237–247, 1993.

[6] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997.

[7] Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 1998.

[8] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, page 60. ACM, 2007.

[9] Simon Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Kevin Hall, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview, 1992.

[10] Peter A. Jonsson and Johan Nordlander. Positive supercompilation for a higher order call-by-value language. In *POPL '09: Proceedings of the 36th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009.

[11] Jan Kort. Deforestation of a raytracer. *Master's thesis, Department of Computer Science, University of Amsterdam, The Netherlands*, 1996.

[12] Duncan Coutts, Roman Leshchinskiy, and Donald Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.

[13] Will Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[14] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

[15] Ilya Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Technical Report Preprint 63, Keldysh Institute of Applied Mathematics, 2009.

[16] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. *Prentice-Hall International Series In Computer Science*, page 415, 1993.

[17] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and gpc. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 485–500, London, UK, 1994. Springer-Verlag.

[18] Peter Sestoft. The structure of a self-applicable partial evaluator. In *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 236–256. Springer Berlin / Heidelberg, 1986.

[19] Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer Berlin / Heidelberg, 2008.

[20] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–268, New York, NY, USA, 1992. ACM.

[21] Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: a programming language for real-time embedded systems. Technical Report CSE-02-002, Oregon Health & Science University, 2002.

[22] Peter A. Jonsson and Johan Nordlander. Strengthening supercompilation for call-by-value languages. In *META 2010*, 2010.

[23] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007.