

Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation

Maximilian Bolingbroke

University of Cambridge
mb566@cam.ac.uk

Simon Peyton Jones

Microsoft Research
simonpj@microsoft.com

Abstract

Supercompilation is a powerful technique for program optimisation and theorem proving. In this paper we describe and evaluate three improvements to the Cambridge Haskell Supercompiler (CHSC). We reduce supercompiled program size by the use of a weak normaliser and aggressive rollback, and we improve the performance of supercompiled programs by heap speculation and generalisation. Our generalisation method is simpler than those in the literature, and is better at generalising computations involving primitive operations such as those on machine integers. We also provide the first comprehensive account of the tag-bag termination mechanism.

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 Algorithms, Performance

1. Overview

Supercompilation is a powerful program transformation technique due to Turchin [1] which can be used to both automatically prove theorems about programs [2] and greatly improve the efficiency with which they execute [3]. Supercompilation is capable of achieving transformations such as deforestation [4], function specialisation and constructor specialisation [5].

Despite its remarkable power, the transformation is simple, principled and fully automatic. Supercompilation is closely related to partial evaluation, but can achieve strictly more optimising transformations [6].

The key contributions of this paper are as follows:

- We briefly describe the structure of our call-by-need supercompiler¹. This improves on prior work [7] in two ways. Firstly, we allow the supercompiler to tie back more often (reducing supercompiler run time and output size) by making use of a weak term normalisation procedure (Section 2.4). Secondly, we give

¹Full source code is available online at <https://github.com/batterseapower/chsc/tree/chsc2>

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.

the first precise account of the tag-bag termination mechanism (Section 2.1).

- We show (Section 3) how to reduce code duplication caused by supercompilation by letting the supercompiler *roll back* to a previous state at certain points.
- We describe a novel approach to term generalisation in a supercompiler that uses tags, which we call the *growing tag heuristic* (Section 4). This generalisation method is easier to implement than standard generalisation techniques.
- Any serious supercompiler for a lazy language must take care to preserve sharing; that is, it must use call-by-need rather than call-by-name. This in turn can lead to a loss of information due to work duplication concerns (Section 5). We describe a new thunk-speculating semantics that can exploit thunks, such as partial applications, for which no work is lost by duplication.
- Finally, we show a natural extension of the growing tag heuristic that allows our supercompiler to better generalise when supercompiling programs that use literals and primitive operations over them (Section 6).

2. Basic supercompilation

The supercompiler is an optimiser that transforms a program to a semantically-equivalent but more efficient form, by aggressive specialisation. The language we supercompile is a conventional untyped, non-strict, higher-order; its syntax is given in Figure 1. We discuss the details in due course, but terms e include the lambda calculus, plus **let** bindings, literals and their primitive operations (primops), and data constructors and **case** expressions.

We begin by reviewing the high-level structure of our supercompiler, which we call CHSC2 (for Cambridge Haskell Supercompiler). CHSC2 builds directly on our previous work, CHSC1 [7], which the reader is urged to consult for further background. As well as introducing our notation and framework, we also describe several significant improvements in CHSC2 compared to CHSC1, and give the first precise account of the tag-bag approach to termination. This section will form a core upon which we will build in subsequent sections when we describe each of the improvements we have made to the basic supercompiler.

The supercompilation algorithm is as follows:

$$sc, sc' :: History \rightarrow State \rightarrow ScpM \ Term$$
$$sc \ hist = memo (sc' \ hist)$$
$$sc' \ hist \ state = \mathbf{case} \ terminate \ hist \ state \ \mathbf{of}$$
$$\quad Stop \quad \rightarrow \ split \ (sc \ hist) \ state$$
$$\quad Continue \ hist' \rightarrow \ split \ (sc \ hist') \ (reduce \ state)$$

In the subsequent sections we will pick apart this definition and explain all the auxiliary functions it makes use of. All the the main

Variables	x, y, z	Primitives	$\otimes ::= +, -, \dots$
Data Constructors	$C ::= True, Just, (:), \dots$		
Literals	$\ell ::= 1, 2, \dots, 'a', 'b', \dots$		
Values			
$v ::=$	$\lambda x. d$	Lambda abstraction	
	ℓ	Literal	
	$C \bar{x}$	Saturated constructed data	
	\underline{x}	Indirection to value	
Terms			
$d ::=$	e^t	Tagged expression	
$e ::=$	x	Variable	
	v	Values	
	$d x$	Application	
	$d \otimes d$	Binary primops	
	$\text{let } x = d \text{ in } d$	Recursive let-binding	
	$\text{case } d \text{ of } \alpha \rightarrow d$	Case decomposition	
Case Alternative			
$\alpha ::=$	ℓ	Literal alternative	
	$C \bar{x}$	Constructor alternative	
States	$S ::= \langle H \mid e \mid K \rangle$		
Heaps	$H ::= \overline{x \mapsto d}$	Stacks	$K ::= \overline{\kappa^t}$
Stack Frames			
$\kappa ::=$	update x	Update frame	
	$\bullet x$	Apply to function value	
	case $\bullet \text{ of } \overline{\alpha \rightarrow d}$	Scrutinise value	
	$\bullet \otimes d$	Apply first value to primop	
	$v^t \otimes \bullet$	Apply second value to primop	

Figure 1: Syntax of the Core language and evaluator

type	$Tag = Int$
type	$Heap = Map Var Term$ -- See H in Figure 1
type	$Stack = [StackFrame]$
data	$UStackFrame = \dots$ -- See κ in Figure 1
type	$StackFrame = (Tag, UStackFrame)$
data	$UTerm = \dots$ -- See e in Figure 1
type	$Term = (Tag, UTerm)$ -- See d in Figure 1
data	$Value = \dots$ -- See v in Figure 1
data	$QA = Question Var \mid Answer Value$
type	$State = (Heap, (Tag, QA), Stack)$ -- Normalised
type	$UState = (Heap, Term, Stack)$ -- Unnormalised
	$freeVars :: State \rightarrow [Var]$
	$rebuild :: State \rightarrow Term$
	$sc :: History \rightarrow State \rightarrow ScpM Term$
	-- The normaliser (Section 2.4)
	$normalise :: UState \rightarrow State$
	-- The evaluator (Section 2.5)
	$step :: State \rightarrow Maybe UState$
	$reduce :: State \rightarrow State$
	-- The splitter (Section 2.7)
	$split :: Monad m \Rightarrow (State \rightarrow m Term)$
	$\rightarrow State \rightarrow m Term$
	-- Termination checking (Section 2.3)
type	$History = [State]$
	$emptyHistory = [] :: History$
data	$TermRes = Stop \mid Continue History$
	$terminate :: History \rightarrow State \rightarrow TermRes$
	-- Memoisation and the $ScpM$ monad (Section 2.6)
	$memo :: (State \rightarrow ScpM Term)$
	$\rightarrow State \rightarrow ScpM Term$

Figure 2: Types used in the basic supercompiler

data types and function signatures are summarised in Figure 2. In particular, the type of sc shows a unique feature of our approach to supercompilation, namely that sc takes as input the *machine states* that are the subject of the reduction rules, rather than *terms* as is more conventional. We review the importance of this choice in Section 2.4.

The main supercompiler sc is built from the following four, mostly independent, subsystems:

1. The *memoiser* (Section 2.6). The call to $memo$ checks whether the input $State$ is equal (modulo renaming) to a $State$ we have already supercompiled, or are in the process of supercompiling. It is this memoisation that “ties the knot” so that a recursive function is not specialised forever. The memoiser records its state in the $ScpM$ monad.
2. The *termination criterion* (Section 2.3). With luck the memoiser will spot an opportunity to tie back, but for some functions – for example, one with an accumulating parameter – each recursive call looks different, so the memoiser will never encounter an identical state. So, if $memo$ does not fire, sc' uses $terminate$ to detect divergence (conservatively, of course). Its $History$ argument allows $terminate$ to accumulate the $States$ it has seen before.
3. The *splitter* (Section 2.7). If the $State$ fails the termination test (the $Stop$ branch), the supercompiler abandons the attempt to

optimise the entire term. Instead, it splits the term into a *residual* “shell” and some (smaller) sub-terms. Then it supercompiles the sub-terms and glues the results back into the shell.

4. The *evaluator* (Section 2.5). If the $State$ passes the termination test (the $Continue$ branch), we call $reduce$ to perform symbolic evaluation of the $State$. This may take many steps, but will eventually halt (it has its own internal termination test). When it does so, we split and residualise just as before.

To get things rolling, the supercompiler is invoked as follows to supercompile an initial input $Term$:

```
supercompile :: Term -> Term
supercompile e = runScpM (sc emptyHistory init_state)
  where init_state = normalise (emptyHeap, e, [])
```

The $normalise$ function will be discussed in Section 2.4.

2.1 Tagged evaluation

A strength of our supercompiler is that it contains at its heart a call-by-need evaluator that is modularly separate from the rest of the supercompiler, and is an absolutely direct implementation of the operational semantics of the language. (In GHC, by contrast, evaluation and optimisation are intimately interwoven.)

It is important to use an *operational* semantics, because the supercompiler must be fastidious about preserving sharing: an “op-

	$\langle H \mid e \mid K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle$	$\begin{aligned} \text{deref}(H[x \mapsto v], \underline{x}) &= \text{deref}(H, v) \\ \text{deref}(H, v) &= v \end{aligned}$
VAR	$\langle H, x \mapsto d \mid x^t \mid K \rangle \rightsquigarrow \langle H \mid d \mid \text{update } x^t, K \rangle$	
UPDATE	$\langle H \mid v^{tv} \mid (\text{update } x)^{tx}, K \rangle \rightsquigarrow \langle H, x \mapsto v^{tv} \mid \underline{x}^{tv} \mid K \rangle$	
APP	$\langle H \mid (d x)^t \mid K \rangle \rightsquigarrow \langle H \mid d \mid \bullet x^t, K \rangle$	
BETA	$\langle H \mid v \mid \bullet x^{tx}, K \rangle \rightsquigarrow \langle H \mid d \mid K \rangle$	$(\lambda x. d)^{t\lambda} = \text{deref}(H, v)$
BETA-N	$\langle H \mid (\lambda x. d)^{t\lambda} \mid \bullet x^{tx}, K \rangle \rightsquigarrow \langle H \mid d \mid K \rangle$	
PRIM	$\langle H \mid (d_1 \otimes d_2)^t \mid K \rangle \rightsquigarrow \langle H \mid d_1 \mid (\bullet \otimes d_2)^t, K \rangle$	
PRIM-L	$\langle H \mid v_1^{tv} \mid (\bullet \otimes d_2)^{t\otimes}, K \rangle \rightsquigarrow \langle H \mid d_2 \mid (v_1^{tv} \otimes \bullet)^{t\otimes}, K \rangle$	
PRIM-R	$\langle H \mid v_2 \mid (v_1 \otimes \bullet)^{t\otimes}, K \rangle \rightsquigarrow \langle H \mid (\otimes(\ell_1, \ell_2))^{t\otimes} \mid K \rangle$	$\ell_i^{t_i} = \text{deref}(H, v_i)$
CASE	$\langle H \mid (\text{case } d_{\text{scrut}} \text{ of } \overline{\alpha \rightarrow d})^t \mid K \rangle \rightsquigarrow \langle H \mid d_{\text{scrut}} \mid (\text{case } \bullet \text{ of } \overline{\alpha \rightarrow d})^t, K \rangle$	
DATA	$\langle H \mid v \mid \text{case } \bullet \text{ of } \{\dots, \mathbf{C}_i \bar{x}_i \rightarrow d_i, \dots\}^{t_2}, K \rangle \rightsquigarrow \langle H \mid d_j \mid K \rangle$	$(\mathbf{C}_j \bar{x}_j)^{t_1} = \text{deref}(H, v)$
LIT	$\langle H \mid v \mid \text{case } \bullet \text{ of } \{\dots, \ell_i \rightarrow d_i, \dots\}^{t_2}, K \rangle \rightsquigarrow \langle H \mid d_j \mid K \rangle$	$\ell_j^{t_j} = \text{deref}(H, v)$
LETREC	$\langle H \mid (\text{let } \overline{x = d} \text{ in } d_{\text{body}})^t \mid K \rangle \rightsquigarrow \langle H, x \mapsto d \mid d_{\text{body}} \mid K \rangle$	

Figure 3: Operational semantics of tagged Core

timeriser” that made programs go more slowly by losing sharing, would not be popular. In particular, the operational semantics implements *call by need* (or lazy evaluation) rather than *call by name*, thereby respecting Haskell’s cost model.

Our Sestoft-style [8] operational semantics are given in Figure 3. The semantics describes transitions from one *machine state* to another. A state $\langle H \mid e \mid K \rangle$ has the following components:

- The *heap*, H , is a finite mapping from variable names to the term to which that variable is bound.
- The *focus term*, e , is the focus of evaluation.
- The *stack*, K , describes the evaluation context of the focus term.

Much of the operational semantics is conventional, but there are three unusual features. The first is the inclusion of *indirections*, to be discussed in Section 2.2. The second unusual feature is the apparent redundancy in the inclusion of BETA-N when we already have BETA. This is necessary to support normalisation, discussed in Section 2.4.

Finally, the heap, focus term, and stack are all *tagged*. A *tagged term* (d in Figure 1) is simply a term e with a tag t , written e^t . Before supercompilation begins, the untagged input program is tagged, once and for all, with a fresh tag at every node. We will write *Tag* for the set of tags thus annotated on the original input term.

Tags are used for two quite separate purposes in CHSC2. First, they are used to drive the termination test (Section 2.3), and second they are used to guide generalisation (Section 4). Neither is entirely new [7, 9], but the contribution of this paper is for the first time to make precise these earlier informal accounts, and to introduce some useful refinements.

The operational semantics of Figure 3 makes explicit exactly how tags are propagated during evaluation. The unbreakable rule is that *no new tags are generated* — it is this invariant that guarantees termination — but that leaves plenty of room for variations of detail. For example, PRIM-R might sensibly use the tag from either argument of the primop to tag the result, and UPDATE could tag the indirection it creates with the update frame tag rather than that of the value. The exact choice can in turn influence both the termination test, and the effectiveness of generalisation (Section 4).

2.2 Indirections

The only non-standard feature of the language is our inclusion of *indirections* in the grammar of values (Figure 1). Indirections are variables that are guaranteed to point to a *value* in the heap, rather than a possibly-unevaluated term. They never occur in input programs, and are only introduced by the UPDATE rule: when the rule inserts the value v^t of x back into the heap H , we continue with an indirection \underline{x} to that value in the focus of evaluation, rather than copying the whole value from the heap into the focus.

Because values may be indirections as well as simple λ s or **data**, reduction rules such as those for β -reduction have to make an auxiliary $\text{deref}(H, v)$ call to “chase pointers” through the heap to the definition of the value in the focus.

The reasons for this revised approach are twofold. Firstly, and more importantly, we found by adding indirections to the rules we could define a more useful normalising subset of the evaluation rules: see Section 2.4 for details on this. Secondly, by creating an indirection rather than copying a value outright, we can avoid duplicating the allocation of that value when we later turn the supercompiled version of the state back into a term. For example, consider this term:

$$\text{let } x = \text{Just } a; y = x \text{ in case } y \text{ of } \text{Just } _ \rightarrow f \ x \ y \\ \text{Nothing} \rightarrow g \ x \ y$$

If the update rule were to copy the value rather than creating an indirection, the supercompiled output would allocate a *Just* twice:

$$\text{let } x = \text{Just } a; y = \text{Just } a \text{ in } f \ x \ y$$

whereas by creating an indirection when we execute the update frame for y , we can get this result instead:

$$\text{let } x = \text{Just } a; y = x \text{ in } f \ x \ y$$

2.3 The termination criterion

The core of the supercompiler’s termination guarantee is provided by a single function, *terminate*:

$$\text{terminate} :: \text{History} \rightarrow \text{State} \rightarrow \text{TermRes} \\ \text{data TermRes} = \text{Stop} \mid \text{Continue History}$$

As the supercompiler proceeds, it builds up an ever-larger *History* of previously-observed *States*. 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, \dots there can be no infinite sequence of calls to *terminate* of this form:

$$\begin{aligned} \text{terminate } h_0 \ s_0 &= \text{Continue } h_1 \\ \text{terminate } h_1 \ s_1 &= \text{Continue } h_2 \\ &\dots \end{aligned}$$

Instead, there will always exist some j such that:

$$\text{terminate } h_j \ s_j = \text{Stop}$$

In Section 2.5 we will see how *reduce* uses *terminate* to ensure that it only performs a bounded number of reduction steps. The same *terminate* function (with a different *History*) is also used *sc*, to ensure that *sc* only recurses a finite number of times, thus ensuring the supercompiler is total².

One way to implement such a termination criterion is by defining a *well-quasi-order* [12]. The relation $\preceq \in S \times S$ is a well-quasi-order (WQO) iff for all infinite sequences of elements of S (s_0, s_1, \dots), it holds that: $\exists i, j. i < j \wedge s_i \preceq s_j$. Given any WQO $\preceq: \text{State} \times \text{State}$, we can implement a sound *terminate* function:

```

type History = [State]
terminate :: History → State → TermRes
terminate prevs here
  | any ( $\preceq$  here) prevs = Stop
  | otherwise = Continue (here : prevs)

```

To implement a sound WQO on *States* we follow Mitchell [9] by extracting a compact summary of the *State*, a bag of tags and comparing those *tag-bags*³:

$$s_1 \preceq s_2 = \text{tagBag } s_1 \preceq_b \text{tagBag } s_2$$

The relation (\preceq) is a WQO on *States* if \preceq_b is a WQO on tag-bags. Again following Mitchell we define \preceq_b as:

$$b_1 \preceq_b b_2 \iff \text{set}(b_1) = \text{set}(b_2) \wedge |b_1| \leq |b_2|$$

Theorem 2.1. \preceq_b is a well-quasi-order.

We can extract a tag-bag from a *State* straightforwardly:

$$\begin{aligned} \text{tagBag} :: \text{State} &\rightarrow \text{TagBag} \\ \text{tagBag } \langle H \mid e^t \mid K \rangle &= \{ \{ 2 * t \mid x \mapsto e^t \in H \} \\ &\cup \{ \{ 3 * t \} \} \\ &\cup \{ \{ 5 * t \mid \kappa^t \in K \} \} \end{aligned}$$

We use $\{ \dots \}$ notation to introduce bag literals and comprehensions with the standard meaning, and overload the set operators \cup and \emptyset to be applicable to bags. Notice carefully that we only collect the *root* tag from each heap binding, the focus term, and stack frame. We do not collect tags from internal nodes. Doing so would make the bags bigger without making the termination test more discriminating because, remembering that tags all derive from the originally-tagged input program, a node tagged t will usually have the same children tags.

We multiply tags by a different constant according to whether they arose from the heap, stack or focus, in order to prevent the termination test from failing immediately just because a reduction step has shuffled syntax between these three locations.

2.4 Normalisation

Our decision to make the supercompiler optimise *States* rather than *Terms* is a consequence of our use of a *State*-based operational semantics, but it pays off elsewhere in the supercompiler

² We do not prove this fact here, but it is a standard technique [10, 11].

³ It is simple matter to memoise the repeated use of *tagBag* on *prevs*.

— particularly in the splitter (Section 2.7), which operates distinctively on each of the three component of the *State*.

Compared to our earlier work in CHSC1, we go further by insisting that all *States* are *normalised* by exhaustively applying all the reduction rules of Figure 3 except for the BETA rule. We still apply BETA-N during normalisation — BETA-N differs from BETA in that it is not allowed to inline the definition of a function from the heap — and it is precisely this inlining that risks divergence.

Theorem 2.2. *The system of reduction rules excluding BETA is strongly normalising.*

Proof. By constructing a suitable measure on states that can be shown to be strictly decreased by every rule. \square

Normalisation has several benefits:

- Because only normalised states are matched, trivial differences between *States* such as the ones described above cannot prevent tieback from occurring (Section 2.6). For example, CHSC1 suffered from increased output program size because not determine that the following *States* all meant the same thing:

$$\begin{aligned} &\langle \text{foldr } \mapsto \dots \mid \text{foldr } \mid \bullet c, \bullet n \rangle \\ &\langle \text{foldr } \mapsto \dots \mid \text{foldr } c \mid \bullet n \rangle \\ &\langle \text{foldr } \mapsto \dots \mid \text{foldr } c \ n \mid \epsilon \rangle \end{aligned}$$

- It reduces the number of times that we have to test the termination criteria in *reduce*, since we only risk non-termination when we perform a non-normalising reduction (see Section 2.5).
- Programs output by our supercompiler will never contain “obvious” undone case discrimination nor manifest β -redexes.

As foreshadowed in Section 2.2, the use of indirections lets us define a more natural normal form. A natural normalising semantics for an indirectionless language is that which has an unrestricted BETA rule but a *restricted* UPDATE rule that cannot duplicate lambdas. As a result, normalised states will often have syntactic λ -values in their focus, to the detriment of readability. In the indirection-based semantics, normalising evaluation can proceed further and continue past the update, stopping only at the point at which the function is actually applied (if any).

Normalised states are always in one of the following three forms:

$$\langle H \mid v \mid \epsilon \rangle \quad \langle H \mid x \mid K \rangle (x \notin H) \quad \langle H \mid \underline{f} \mid \bullet x^t, K \rangle$$

We therefore define the *State* type to be the type of *normalised* states, and represent the fact that the focus will only ever be a variable or a value by introducing a new sum type *QA* (see Figure 2). The *normalise* function allows a non-normalised state (*UState*) to be injected into the *State* data type in the obvious way:

$$\text{normalise} :: \text{UState} \rightarrow \text{State}$$

2.5 Reduction

Normalisation performs lots of reduction, but the supercompiler still needs to perform some potentially non-normalising reduction steps in order to make good progress. We do such reductions with a function *step* that it implements the BETA reduction rule (Figure 3).

$$\text{step} :: \text{State} \rightarrow \text{Maybe } \text{UState}$$

If BETA fires, *step* returns an un-normalised state, *UState*; otherwise it returns *Nothing*. Typically, normalisation is then applied to the reduced result.

We can use *step* and *normalise* together to build a terminating multi-step evaluator, which in turn is called by the main supercompiler *sc'*:

$reduce :: State \rightarrow State$
 $reduce = go\ emptyHistory$

where

$go\ hist\ state = \mathbf{case}\ terminate\ hist\ state\ \mathbf{of}$

$Stop \rightarrow state$

$Continue\ hist' \rightarrow \mathbf{case}\ step\ state\ \mathbf{of}$

$Nothing \rightarrow state$

$Just\ us \rightarrow go\ hist'\ (normalise\ us)$

The totality of *reduce* is achieved using the *terminate* function from Section 2.3. If *terminate* reports that evaluation appears to be diverging, *reduce* immediately returns. As a result, the *State* triple (h, qa, k) returned by *reduce* might not be fully reduced (though it will of course be normalised in the sense of Section 2.4) — in particular, it might be the case that $qa = \underline{f}$ where f is bound by h .

2.6 Memoisation

The purpose of the memoisation function, *memo*, is to ensure that we reuse the results of supercompiling a *State* if we come to supercompile an equivalent *State* later on.

We achieve this by using the *ScpM* state monad: whenever we supercompile a new *State* we give it a fresh name of the form h_n for some n and record the *State* and name in the monad as a *promise*. The supercompiled version of that *State* will be λ -abstracted over its free variables and bound to that name at the top level of the output program. Now if we are ever asked to supercompile a later *State* equivalent to that earlier one (up to renaming) we *tie back* by just looking in the promises and producing a call to the name of the earlier state, applied to appropriate arguments. More detail is given in our earlier paper [7].

2.7 The splitter

The job of the splitter is to complete the process of supercompiling a *State* whose reduction is stuck, either because of a lack of information (e.g. if the *State* is blocked on a free variable), or because the termination criterion is preventing us from reducing that *State* further. The splitter has the following type signature:

$$split :: Monad\ m \Rightarrow (State \rightarrow m\ Term) \\ \rightarrow State \rightarrow m\ Term$$

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 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*:

$$split_s = return\ (rebuild\ s)$$

(*rebuild* turns a *State* back into a *Term*; Figure 2.) 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*. Space limitations preclude us from discussing *split* in great detail, but further details can be found in our previous paper [7] or the published source code.

To explain a little about how *split* works, we first introduce a notational device similar to that of Mitchell [9] 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 *States*:

$$\mathbf{let}\ x = \langle \epsilon \mid 1 \mid \epsilon \rangle ; xs = \langle \epsilon \mid map\ (const\ 1)\ ys \mid \epsilon \rangle \\ \mathbf{in}\ x : xs$$

You should read this in the following way:

- The part of the term outside the $\langle \text{state brackets} \rangle$ is the *residual* code that will form part of the output program.
- In contrast, the brackets enclose sub-*States* that are fed to *opt* for further supercompilation.

Before *split* returns, the result of supercompiling each sub-*State* is pasted into the correct position in the residual code. So the actual end result of such a supercompilation run might be something like:

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

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:

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

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* is when evaluation of a **case** expression gets stuck. As an example, consider the following (stuck) input to *split*:

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

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

$$\mathbf{case}\ x\ \mathbf{of}\ True \rightarrow \langle \epsilon \mid 1 \mid \epsilon \rangle \\ False \rightarrow \langle \epsilon \mid 2 \mid \epsilon \rangle + \langle \epsilon \mid 3 \mid \epsilon \rangle$$

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:

$$\mathbf{case}\ x\ \mathbf{of}\ True \rightarrow \langle \epsilon \mid 1 \mid (\bullet + 3) \rangle \\ False \rightarrow \langle \epsilon \mid 2 \mid (\bullet + 3) \rangle$$

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

This is one of the places where the decision to have the supercompiler work with *States* rather than *Terms* 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.

2.8 A complete example

Here is an example of the supercompiler in action. Consider the function *map*, whose tagged definition is as follows:

$$map\ f\ xs = (\mathbf{case}\ xs^{t_1}\ \mathbf{of} \\ [] \rightarrow [] \\ (y : ys) \rightarrow ((f\ y)^{t_2} : ((map^{t_3}\ f)^{t_4}\ ys)^{t_5})^{t_6})^{t_7}$$

Now, suppose we supercompile the call $((map^{t_a}\ not)^{t_b}\ xs)^{t_c}$, which inverts every element in a list of booleans, xs . Remember (Section 2) that *supercompile* normalises the expression before

giving it to sc . Normalisation will evaluate the term until it gets stuck, which is nearly immediate, because it needs to inline map . So the initial $State$ given to sc is this:

$$S_1 = \langle map \mapsto (\dots)^{t_9}, not \mapsto (\dots)^{t_8} \mid \underline{map}^{t_9} \mid \bullet not^{t_b}, \bullet xs^{t_c} \rangle$$

As this is the first invocation of sc , there is no way we can tie back, so sc' is called. The history is empty, so the termination check passes — after extending the history with the tag-bag $\{3 * t_9, 5 * t_b, 5 * t_c, 2 * t_9, 2 * t_8\}$ — and so sc' calls $reduce$ which evaluates the state (including inlining; see Section 2.5) until it gets stuck because it has no binding for xs :

$$\langle map \mapsto (\dots)^{t_9}, not \mapsto (\dots)^{t_8} \mid xs^{t_1} \mid \kappa^{t_\tau} \rangle$$

where κ is the stack frame for the case expression:

$$\kappa = \mathbf{case}\bullet\mathbf{of} \left\{ \begin{array}{l} [] \rightarrow [] \\ (y : ys) \rightarrow ((not\ y)^{t_2} : ((map^{t_3}\ not)^{t_4}\ ys)^{t_5})^{t_6} \end{array} \right\}$$

Now $split$ residualises part of the reduced $State$ and recursively supercompiles the two branches of the case. We concentrate on the $(:)$ branch. Once again the $State$ constructed for this branch is normalised before being passed to the recursive invocation of sc . The normalised state looks like this:

$$\left\langle \begin{array}{l} map \mapsto (\dots)^{t_9}, not \mapsto (\dots)^{t_8}, \\ z \mapsto (not\ y)^{t_2}, zs \mapsto ((map^{t_3}\ not)^{t_4}\ ys)^{t_5} \end{array} \mid (z : zs)^{t_6} \mid \epsilon \right\rangle$$

We cannot tie back at this point, and the tag-bag for this state, $\{2 * t_2, 2 * t_5, 3 * t_6, 2 * t_9, 2 * t_8\}$, is distinct (as a set) from the previous $State$, and so supercompilation proceeds in the $Continue$ branch of sc' by splitting the $State$ ($reduce$ is the identity function on this $State$ as it is already a value). Both the head z and tail zs of the output list are recursively supercompiled, but we focus on the tail. Normalising gives:

$$S_3 = \langle map \mapsto (\dots)^{t_9}, not \mapsto (\dots)^{t_8} \mid \underline{map}^{t_9} \mid \bullet not^{t_4}, \bullet xs^{t_5} \rangle$$

When sc is invoked on S_3 , it notices that it has already supercompiled the α -equivalent state S_1 , and so it returns immediately with a $Term$ that just invokes the corresponding h -function, $h1$.

The final output program (including the $h4$ and $h5$ functions generated by invocations of sc that we have elided) is thus:

$$\begin{aligned} h1\ xs &= \mathbf{case}\ xs\ \mathbf{of}\ [] \quad \rightarrow h4 \\ &\quad (y : ys) \rightarrow h2\ y\ ys \\ h2\ y\ ys &= h5\ y : h1\ ys \\ h4 &= [] \\ h5\ y &= not\ y \end{aligned}$$

After inlining functions that are called exactly once we recognise $h1$ as a version of map specialised for not as its first argument.

3. Rollback

Suppose that we apply $reduce$ to the following term:

$$\mathbf{let}\ f\ x = x + f\ x\ \mathbf{in}\ f\ 2$$

Evaluation yields a sequence of terms $f\ 2, 2 + f\ 2, 2 + 2 + f\ 2$, and so does not terminate. Therefore, $terminate$ will eventually say $Stop$, and $reduce$ (as defined in Section 2.5) will return whatever term it has reached, say $2 + 2 + f\ 2$. But since we have detected probable divergence it might be better to discard the fruitless work and *roll back* to an earlier term that has not grown so much.

Because the termination criteria is used in two ways in the supercompiler, there are two opportunities to introduce rollback — we not only add it to $reduce$, but to sc as well.

3.1 Rolling back reduction

The most straightforward change is to the $reduce$ function (Section 2.5). It would be possible for $reduce$ to revert all the way to its

original input term in the event of divergence, but that risks discarding *useful* computation along with the bloat. For example, suppose the body of the \mathbf{let} in the example above was $id\ (f\ 2)$ where id is the identity function. Then it would be a pity to discard the work of reducing the call to id .

A more sophisticated approach is to modify the termination test API, so that when reporting $Stop$ it also yields information recorded along with the earlier $State$ that was \leq the new one:

```
type History a = [(State, a)]
emptyHistory :: History a
data TermRes a = Stop a | Continue (History a)
terminate :: History a → State → a → TermRes a
terminate hist here here_extra
  | prev_extra: _ ← [ prev_extra
                    | (prev, prev_extra) ← hist
                    , prev ≤ here ]
  = Stop prev_extra
  | otherwise
  = Continue ((here, here_extra) : hist)
```

To allow $reduce$ to rollback, we can now use the extra data field in the $History$ to store the current $State$ whenever the termination criteria was tested⁴. Should we be forced to $Stop$ reduction, that stored $State$ is returned instead of the latest (more-reduced) version. The new code may be compared with that in Section 2.5:

```
reduce :: State → State
reduce = go emptyHistory
where
  go hist state = case terminate hist state state of
    Stop old_state → old_state
    Continue hist' → case step state of
      Nothing → state
      Just us → go hist' (normalise us)
```

A simple example of where this makes a difference is when using $reduce$ on a term such as this:

```
let loop xs = loop (1 : xs); id x = x
in id (loop [])
```

CHSC1 would produce a $State$ such as this one:

```
let loop xs = loop (1 : xs)
  xs0 = []; xs1 = (:) 1 xs0
  xs2 = (:) 1 xs1; xs3 = (:) 1 xs2
in loop xs3
```

CHSC2 instead rolls back to an earlier $State$ where we have duplicated less code:⁵

```
let loop xs = loop (1 : xs); xs0 = []
  xs1 = (:) 1 xs0; xs2 = (:) 1 xs1
in loop xs2
```

The $reduce$ function with rollback is, pleasingly, idempotent.

3.2 Rolling back driving

The other use of $terminate$ occurs in the sc function itself, where it controls how deeply nested recursive invocations of sc can become. We would also like to roll back here, but doing so is complicated

⁴ It may seem redundant to store the $State$ twice in each $History$ entry, but this design is chosen for uniformity with Section 3.2

⁵ CHSC2 still unrolls the call to $loop$ twice, whereas one unrolling might seem more reasonable. This happens because the call to $loop$ in the body of the \mathbf{let} is assigned a different tag to the occurrence of $loop$ in the body of $loop$ itself.

by the monadic structure of sc – we would like to roll back the monad-carried information as well.

The easiest way to implement this is by making $ScpM$ an exception monad, in which rollback is triggered by throwing an exception. However, rollback should not revert to the immediately-enclosing invocation of sc but rather to the invocation that processed the $State$ that is \leq the current state. So we need to throw an exception that will only be caught by the “right” handler. An elegant way to express this idea is with a single new primitive in the $ScpM$ monad:

```

type Throw c =  $\forall b. c \rightarrow ScpM b$ 
catchScpM :: (Throw c  $\rightarrow ScpM a$ ) -- Action to try
            $\rightarrow (c \rightarrow ScpM a)$  -- Handler
            $\rightarrow ScpM a$ 

```

The second argument is the easy one: it is the exception handler, invoked if the first argument throws an exception. The first argument is the action to run in the scope of the exception handler, but it needs to know how to throw an exception to this particular invocation of $catchScpM$. So $catchScpM$ applies its first argument to a freshly-minted “how to throw” function. This latter function takes a value of the type expected by the handler, c in this signature, and throws the exception. This signature allows the code that raises the exception to pass a value of type c to the handler, to communicate some information about the failure.

Given $catchScpM$, we can implement a version of sc with rollback. We make use of the same enhanced $terminate$ function, but this time the “extra information” passed to $terminate$ and returned by $Stop$ is the “how to throw function”.

```

type ThrowTerm = ()  $\rightarrow ScpM Term$ 
sc' :: History ThrowTerm  $\rightarrow State \rightarrow ScpM Term$ 
sc' hist state
  = ( $\lambda throw \rightarrow check\ hist\ throw$ ) 'catchScpM'
  ( $\lambda () \rightarrow split\ (sc\ hist)\ state$ )
where
  check hist throw = case terminate hist state throw of
    Stop old_throw  $\rightarrow old\_throw\ ()$ 
    Continue hist'  $\rightarrow split\ (sc\ hist')\ (reduce\ state)$ 

```

If we are forced to terminate (the $Stop$ branch), then instead of continuing from the current $State$ (which triggered the termination condition), we raise an exception using the exception raiser stored with the $State$ which “blew the whistle”. When resuming execution from an exception handler, we know that supercompiling the state associated with that handler eventually caused the termination criteria to fire. Therefore, we act as if the state had failed the termination test and do not reduce it before splitting. It is remarkable how little the structure of the supercompiler is disturbed by this change.

For now, the “exception type” c in the type of $catchScpM$ is $()$. However, we will need to instantiate it with a more interesting type when we come to consider the generalisation feature of CHSC2 in Section 4.

Note that the $History$ at the point we raise an exception (with old_throw) may be longer than the $History$ at the point we roll back to. In fact, it is not necessary to preserve this longer history when we rollback – we can make do with the shorter history at the point the exception handler was installed. We claim that this does not affect termination of the supercompiler.

One interesting question is what should happen to the $ScpM$ -carried information when we rollback. In particular, in between the time the exception handler was installed and when an exception is raised we may have completed supercompilation of some new h-functions – what happens to those when we roll back?. One strategy

Test	SC ^a	Cmp. ^b	Run ^c	Mem. ^d	Size ^e
digitsofe2	-6%	+1%	+0%	+0%	-8%
rfib	-1%	+0%	+0%	-1%	+0%
tak	-3%	+1%	+64%	+38%	+39%
treeflip	+1%	+0%	-4%	-1%	+0%
Average	-1%	+0%	+4%	+3%	+2%
Maximum	+1%	+1%	+64%	+38%	+39%
Minimum	-7%	-1%	-4%	-1%	-8%

^a Supercompilation time change when feature enabled
^b GHC compile time change when feature enabled
^c Program runtime change when feature enabled
^d Runtime allocation change when feature enabled
^e Code size change (in AST nodes) when feature enabled

Figure 4: The effect of enabling $reduce$ -rollback

would be to discard them, on the basis that since we have rolled back they will not necessarily be used. However, we found that in practice these specialisations were often useful even after rolling back – retaining generated h-functions caused the supercompilation time of the digitsofe2 benchmark to decrease by 85%.

For this reason, our supercompiler retains as many h-functions as possible when rolling back. One subtlety is that in between the point at which the exception handler is installed and that at which the exception is raised we may have made some new promises that have not yet been fulfilled. By rolling back, we make it such that these promises will never be fulfilled – so we *do* have to discard any h-functions (and their dependents) that have tied back on the basis of those promises. All of this jiggery-pokery is neatly hidden away inside the implementation of $catchScpM$.

3.3 Evaluating rollback

We tested the effects of modifying the supercompiler with both forms of rollback by observing the impact of enabling them when supercompiling the benchmark suite we will use for all performance tests in this paper:

- Seven examples from the nofib “imaginary” benchmark suite [13] (bernouilli, digitsofe2, exp3_8, primes, rfib, tak and x2n1)
- Five standard example programs used in previous work on supercompilation and deforestation [3, 4, 9, 14] (append, factorial, raytracer, sumtree, treeflip)
- One benchmark (sumsquare) from work on stream fusion [15].

Our supercompiler (CHSC) is implemented as a preprocessor. All benchmarks are performed compiling the supercompiled program with GHC at the highest – $O2$ optimisation level.

To evaluate the rollback feature, we investigate how the performance of supercompiled programs changes if we disable rollback in the final version of our supercompiler (described in Section 7).

The effects of enabling $reduce$ -rollback are shown in Figure 4. We only show those few programs for which enabling the feature made a difference. This small effect an expected result, because $reduce$ is stopped by the termination criteria only rarely – it is much more common that it stops because of a lack of information. Overall the effect is minimal.

As expected, we do see a reduction in the size of the digitsofe2 output program. Perhaps more surprising is the fact that for the tak benchmark the size of the output program was actually *increased* by enabling $reduce$ -rollback. This change is exaggerated by the small size of tak, and occurs because enabling rollback changes the order in which bindings are generalised (Section 4). The change in heap usage occurs because this new generalisation choice just happens to produce an output program less amenable for analysis by GHC’s strictness analyser, *not* because the supercompiler is deforesting less.

Test	SC	Cmp.	Run	Mem.	Size
append	-1%	+0%	+0%	+0%	+0%
bernoulli	+5%	-1%	-1%	+0%	-3%
digitsofe2	+165%	+8%	-4%	+0%	+83%
exp3.8	+2%	-4%	+4%	+0%	-59%
factorial	-1%	-1%	-3%	+0%	+0%
primes	-7%	+1%	+1%	+0%	-19%
raytracer	+0%	-1%	+0%	+0%	+0%
rfib	-1%	-1%	-1%	+0%	+0%
sumsquare	-34%	-4%	-82%	-100%	-64%
sumtree	-7%	-1%	+11%	-1%	-37%
tak	-28%	-6%	+22%	-100%	-50%
treeflip	-8%	-1%	+25%	-1%	-37%
x2n1	-4%	+0%	-1%	+0%	-22%
Average	+6%	-1%	-2%	-16%	-16%
Maximum	+165%	+8%	+25%	+0%	+83%
Minimum	-34%	-6%	-82%	-100%	-64%

Figure 5: The effect of enabling *sc*-rollback

Figure 5 shows the results of enabling *sc*-rollback in the supercompiler. The effect here is much more pronounced, failing to make an impact on results only for the *append*, *factorial*, *raytracer* and *rfib* benchmarks – which are precisely the three benchmarks that supercompiled cleanly without the *sc* termination test failing. Overall results are very encouraging: we reduce the size of the output programs by 16% on average without adverse effect on memory usage or runtime of output programs.

Indeed, we find that *sc*-rollback tends to have a positive effect on optimisation, despite it “undoing” some of the work of the supercompiler. One reason this happens is because the *History* is rolled back as well – giving us more opportunity for useful specialisation (e.g. in *sumsquare* and *treeflip*) than if we had continued with the (larger) history at the point we rolled back from. We also see evidence that this allows us to generate lots of new specialisations that are themselves eventually rolled back, increasing supercompilation time without corresponding increases in output size (e.g. *digitsofe2*).

One effect of *sc*-rollback is that loops are peeled and unrolled less. However, we would anyway prefer to avoid this sort of optimisation in the supercompiler because it is better done by a lower-level optimisation pass that has access to information about characteristics of the execution hardware.

4. Generalisation by growing tags

A shortcoming of CHSC1 was that it did not make use of the standard technique of *generalisation* to guess good induction hypotheses. Generalisation is an important heuristic for optimising programs whose supercompilation generates sequences of *States* to supercompile that do not tie back – such as those programs that make use of accumulating parameters.

4.1 The problem

An example where generalisation helps optimisation is *foldl*:

$$\text{foldl } c \ n \ xs = (\text{case } xs^{t_1} \text{ of} \\ \begin{array}{l} [] \rightarrow n \\ (y : ys) \rightarrow (((\text{foldl}^{t_2} \ c) \ t_3 \ (c \ n \ y)^{t_4}) \ t_5 \ ys)^{t_6}) \ t_7 \end{array})$$

Now, suppose we supercompile the call $((\text{foldl}^{t_a} \ (+)^{t_b} \ n)^{t_c} \ xs)^{t_d}$, which computes the sum of the elements of *xs*. Remember (Section 2) that *supercompile* normalises the expression before giving it to *sc*. Normalisation will evaluate the term until it gets stuck, which in this case happens nearly immediately because it needs to inline *foldl*. So the initial *State* given to *sc* is this:

$$S_1 = \langle H \mid \text{foldl}^{t_s} \mid (\bullet(+))^{t_b}, (\bullet n)^{t_c}, (\bullet xs)^{t_d} \rangle$$

(In each term in this section we elide the shared heap $H = \text{foldl} \mapsto (\dots)^{t_8}, (+) \mapsto (\dots)^{t_9}$.) Now, *sc* calls *reduce* which inlines *foldl* and then gets stuck because it has no binding for *xs*:

$$\langle H \mid xs^{t_1} \mid \kappa^{t_7} : \epsilon \rangle$$

where κ is the stack frame for the case expression:

$$\kappa = \text{case} \bullet \text{of} \left\{ \begin{array}{l} [] \rightarrow n \\ (y : ys) \rightarrow (((\text{foldl}^{t_2} \ (+)) \ t_3 \ (n + y)^{t_4}) \ t_5 \ ys)^{t_6} \end{array} \right\}$$

Now *split* residualises the *State* and recursively supercompiles the call to *foldl* in the case branch. Once again this *State* is normalised before being passed to the recursive invocation of *sc*:

$$S_2 = \langle H, n1 \mapsto (n + y)^{t_4} \mid \text{foldl}^{t_s} \mid (\bullet(+))^{t_3}, (\bullet n1)^{t_5}, (\bullet ys)^{t_6} \rangle$$

The tag-bag for S_1 is $\{\{5 * t_b, 5 * t_c, 5 * t_d, 3 * t_8, 2 * t_8, 2 * t_9\}\}$ and for S_2 is $\{\{5 * t_3, 2 * t_4, 5 * t_5, 5 * t_6, 3 * t_8, 2 * t_8, 2 * t_9\}\}$, so the termination criterion does not fire. We again *reduce*, and recursively supercompile the call to *foldl* thus revealed. Normalising once more gives,

$$S_3 = \langle H_3 \mid \text{foldl}^{t_s} \mid (\bullet(+))^{t_3}, (\bullet n2)^{t_5}, (\bullet ys1)^{t_6} \rangle \\ H_3 = H, n1 \mapsto (n + y)^{t_4}, n2 \mapsto (n1 + y1)^{t_4}$$

(We must of course freshen the variables bound by the **case**, so have renamed the variables *y* and *ys* bound by the most recent **case** stack frame to *y1* and *ys1* respectively.)

The tag-bag for S_3 is $\{\{5 * t_3, 2 * t_4, 2 * t_4, 5 * t_5, 5 * t_6, 3 * t_8, 2 * t_8, 2 * t_9\}\}$. Considered as a set, this has the same tags as the tag-bag for S_2 , but a greater multiplicity for t_4 – the “growing tag” – so the termination test fires.

At this point, our standard supercompiler (without rollback) would finish up by splitting as follows:

$$\text{let } (+) = \langle (+) \mapsto \dots \mid (+) \mid \epsilon \rangle \\ n2 = \langle (+) \mapsto \dots, n1 \mapsto n + y \mid n1 + y1 \mid \epsilon \rangle \\ \text{in } \langle \text{foldl} \mapsto \dots \mid \text{foldl} \mid \epsilon \rangle \ (+) \ n2$$

This is hopeless: we end up merely creating an exact copy of *foldl*, called after one peel of the loop. What we want is to *specialise* wrt the argument that is not changing, namely $(+)$, but *parameterise* over the argument that *is* changing, the accumulator *n*.

4.2 Our solution

The problem of how to continue when the supercompiler termination criteria fails is well known and is solved by the choice of some *generalisation* method [10, 16]. The goal of generalisation is to use the specialisations generated thus far to infer a “more general” specialisation that subsumes both of them.

Almost all supercompilers implement generalisation by computing the so-called *most specific generalisation* (MSG) between a term and that earlier term into which it embeds. However:

- The MSG is computed using terms that do not contain **let** bindings, which is a problem for a call-by-need supercompiler that needs to preserve **let** bindings in order to prevent loss of sharing. It is not totally clear how to extend the MSG to terms containing **let**.
- The MSG is usually only guaranteed to be non-trivial between terms that are homomorphically embedded. However, we use tag-bags instead of the homeomorphic embedding — so trying to use the MSG is unlikely to give good results.

Instead, we adopt a new approach, which we call the *growing-tags heuristic*. This heuristic is particularly simple, but is as effective as any other that we have tried. The idea is that when the termination test fails there is usually at least one tag that can be “blamed” for this, in the sense that the number of occurrences of that tag has

grown since we supercompiled the prior state. In our *foldl* example, there is indeed such a tag — t_4 . We can use the set of growing tags to generalise the *State* being supercompiled by *residualising any syntax tagged with the growing tag(s)* and then continuing. In our example, when the termination test fails on S_3 , we would residualise the heap bindings for $n1$ and $n2$ because they are tagged (at the root) with t_4 , and recursively supercompile the sub-terms as follows:

```
let n1 = ⟨(+) ↦ ... | n + y | ε⟩
    n2 = ⟨(+) ↦ ... | n1 + y1 | ε⟩
in ⟨(+) ↦ ..., foldl ↦ ... | foldl | • (+), • n2⟩
```

Now, since the accumulator $n2$ has been residualised, the recursively supercompiled *State* in the body of the *let* can immediately tie back to the h function for S_1 , and we get a loop that implements *foldl*. The resulting loop has been peeled once (because the tags of the initial call are different from the tags of subsequent ones), and unrolled twice (because it took two *foldl* inlinings before we could spot the pattern of growth).

This technique combines very nicely with rollback — with rollback enabled, we can simply force residualisation of any syntax in the *older State* that is tagged with a growing tag. So for our *foldl* example, failure of the termination test would cause us to roll-back to S_2 , at which point we would notice that the $n + y$ thunk is tagged with the growing tag t_4 and thus residualise it. The resulting loop will still have been peeled once, but will not have been unrolled.

In general, it might be the case that no tag is growing with respect to the previous tag bag, which happens if and only if the latest tag-bag is exactly equal to the older one. In this case we are forced to split using the standard splitting mechanism used by CHSC1. This is analogous to the situation that occurs with a MSG-based generalisation heuristic where two terms have a trivial MSG.

The plumbing needed to implement this generalisation method can be added to the basic supercompiler in a very straightforward manner. We can replace the definition of sc' from Section 3.2 with the following:

```
type Growing = Set Tag
type ThrowTerm = State → ScpM Term

sc' hist state
= (λthrow → check hist throw) 'catchScpM'
  (λstate' → case findGrowing state state' of
    Nothing → split (sc hist) state
    Just growing → generalise growing (sc hist) state)

where
  check hist throw = case terminate hist state throw of
    Stop old_throw → old_throw state
    Continue hist' → split (sc hist') (reduce state)
```

Note that we instantiated the “exception type” parameter c of *catchScpM* with *ThrowTerm* (c.f. Section 3.2).

To make use of the growing-tags information, we introduce *generalise*, a function similar to *split*, but which only residualises those parts of the input *State* that are marked with a growing tag (and anything that those residualised portions transitively depend on). In our implementation, *generalise* shares almost all of its code with *split*.

```
generalise :: Growing
           → (State → ScpM Term)
           → State → ScpM Term
```

Finally we can compute the growing tags themselves:

```
findGrowing :: State → State → Maybe Growing
findGrowing s1 s2
| setNull g = Nothing
```

Test	SC	Cmp.	Run	Mem.	Size
append	-1%	-8%	+0%	+0%	+0%
bernoulli	-2%	-1%	-2%	-1%	-9%
digitsofe2	+110%	+11%	-41%	-33%	+119%
exp3_8	-64%	-4%	+0%	+0%	-49%
factorial	-1%	+0%	+0%	+0%	+0%
primes	+6%	-2%	-27%	-2%	-47%
raytracer	+0%	+0%	+0%	+0%	+0%
rfib	-2%	+0%	-1%	-1%	+0%
sumsquare	+2%	-1%	-82%	-100%	-7%
sumtree	-49%	-2%	-88%	-100%	-55%
tak	+0%	+2%	-47%	-100%	+74%
treeflip	-66%	-2%	-91%	-100%	-62%
x2n1	-38%	-2%	+5%	-84%	-45%
Average	-8%	-1%	-29%	-40%	-6%
Maximum	+110%	+11%	+5%	+0%	+119%
Minimum	-66%	-8%	-91%	-100%	-62%

Figure 6: The effect of enabling generalisation

```
| otherwise = Just g
where g = bagToSet $ tagBag s2 'bagMinus' tagBag s1
```

In practice, there will sometimes be several growing tags, in which case supercompilation can continue if we residualise syntax tagged with *any* non-empty subset of the growing tags. The choice of exactly which subset to use — a process we call *pruning* — leaves some scope for tuning of the generalisation: we have found that a good heuristic is to residualise those tags that originate from stack frames in preference to those on any other piece of syntax.

4.3 Evaluating generalisation

The effects of enabling generalisation are recorded in Figure 6. The results are very good: supercompilation time, memory usage and output size almost always fall — indeed, no benchmark allocates more after this change. The size of *digitsofe2* increases significantly due to extra specialisation, but it pays off in performance terms. These results underscore the importance of the choice of generalisation heuristic in supercompilation.

5. Speculative evaluation

A call-by-need supercompiler such as CHSC will unavoidably sometimes find itself discarding information from the *Heap* in *split*. This is because some heap bindings will both be shared and expensive to compute, and so we cannot risk duplicating the work they do. A concrete example of this would be a program such as:

```
let b = odd unk
in (if b then x else y, if b then y else x)
```

If we were free to duplicate the b binding into each component of the pair, we could fuse the consumption of the *Bool* result by the *if* expressions into *odd*, hence deforesting the intermediate boolean values. However, doing this would cause the *unk* number to be traversed by *odd* twice. To prevent such problems, *split* forces residualisation of any heap bindings that are not syntactic values.

However, the check for something being a syntactic value is too strict — many things are in fact safe to duplicate even though they are not values. A simple example is a partial application:

```
let f = λx y. x + y; g = f 1
in (g 2, g 3)
```

CHSC1 would produce the following output for this program:

```
let g = λy. 1 + y in (g 2, g 3)
```

The f call has been inlined, but the two calls to g have not been. This is because the g binding appears to be expensive (because

f 1 is an application, not a value) and hence is not duplicated into the components of the pair. Of course, after supercompiling g we discover that it actually evaluates to a value – but we discover that fact too late to make use of it.

5.1 Our solution

In contrast, CHSC2 uses *let-speculation* to preemptively discover that g is a value. After sc calls *reduce*, an auxiliary function *speculate* attempts to reduce any non-values in the resulting *Heap* to values by invoking *reduce* on them. If *reduce* reaches a value then the original binding is replaced with the new form, but if *reduce* fails to reach a value the original binding is retained.

Cheap expressions that are “close to” values (like f 1 in our example) will be replaced with syntactic values by the speculator, which allows *split* to residualise less and propagate more information downwards. The use of this technique means that unlike Mitchell [9], we do not require a separate arity analysis stage to supercompile partial applications well. Furthermore, this technique automatically achieves useful *let-floating*, and also allows fully-applied function calls and primop applications to be duplicated if those calls quickly reach values⁶.

Using *reduce* on some *Heap* binding may itself give rise to further heap bindings requiring speculation. For example, if we speculate the expression `let x = 1 + 1 in Just x`, after reduction the outgoing heap will contain a binding for the term $1 + 1$.

Naturally, we wish to speculate such bindings recursively, but implemented naïvely this creates a new source of non-termination — for example, consider speculating the expression `enumFrom 1`. The *reduce* call will reach a value $1 : ys$ with new heap bindings $y \mapsto 1 + 1$ and $ys \mapsto \text{enumFrom } y$. If we speculate the ys binding in turn we generate the new bindings $z \mapsto y + 1$ and $zs \mapsto \text{enumFrom } z$. We are in a loop.

To head off the resultant non-termination, we use the tag-bag termination test to check each heap binding before speculating it. If the test fails, we do not reduce, and instead roll the speculator back to point at which it was considering speculating the heap binding which blew the whistle, and retrospectively prevent reduction of that binding. This rollback is implemented by an exception-raising mechanism much the same way as described in Section 3.

An alternative to doing rollback would be to simply cease speculation at the point the termination test fails. However, doing so leads to undesirable unrolling as speculating our `enumFrom 1` example from earlier will cause it to be unrolled twice. With rollback we detect the divergence and undo the peeling of `enumFrom`.

Rollback is also important because it ensures that *speculate* is idempotent – this is an important property, as bindings that speculation has failed to reduce to a value may be speculated again in a recursive invocation of *sc* – and we certainly don’t want to do things like unroll an `enumFrom` call once for every time *sc* recurses⁷.

5.2 Evaluating speculation

Figure 7 shows the results of enabling speculation for those benchmarks where it made a difference. The results are modest because our benchmark suite contains few programs that make non-linear use of partial applications and other expressions that are “almost values” — but the added information propagation does cause a big improvement when it happens.

⁶ Speculation of saturated applications is a particularly important feature if (like CHSC) your supercompiler introduces data constructor wrappers to deal with partial application of constructors in the input language

⁷ To improve efficiency, our real implementation keeps track of the names of previously-speculated bindings to avoid speculating them again

Test	SC	Cmp.	Run	Mem.	Size
bernoulli	+77%	+1%	+0%	+0%	+27%
digitsofe2	+208%	+16%	+1%	+0%	+146%
exp3_8	+17%	+1%	+0%	+0%	-2%
primes	+3%	-1%	-9%	+10%	+1%
rfib	-29%	-2%	+0%	-1%	+0%
sumsquare	-11%	-1%	-94%	-100%	-20%
sumtree	-28%	+0%	-1%	+0%	+5%
tak	-3%	-1%	-1%	-1%	+0%
treeflip	-28%	+0%	-3%	+0%	+6%
x2n1	-31%	-3%	+19%	-84%	-45%
Average	+14%	+1%	-7%	-14%	+9%
Maximum	+208%	+16%	+19%	+10%	+146%
Minimum	-31%	-3%	-94%	-100%	-45%

Figure 7: The effect of enabling speculation

The regression in the memory usage of `primes` is once again caused by an adverse interaction with GHC’s strictness analyser. The growth of `digitsofe2` happens because speculation of a heap binding prevents the memoiser from seeing that `tieback` is possible, which in turn necessitates the generation of a large number of duplicate specialisations.

6. Generalisation and primops

Primops shrink syntax trees, discarding tags as they do so. It turns out that discarding these tags makes it harder for the growing-tag generaliser to do a good job. A simple example of the problem occurs when supercompiling a version of `length` defined with the non-strict `foldl` defined in Section 4. Consider this term:

$$\text{let } next - n = n + 1; z = 0^{t_e} \text{ in } ((\text{foldl}^{t_a} \text{ next}^{t_b} z)^{t_c} xs)^{t_d}$$

Supercompiling, we see the following sequence of normalised *States*:

$$\begin{aligned}
 S_1 &= \langle H, z \mapsto 0^{t_e} \mid \text{foldl}^{t_s} \mid (\bullet \text{ next})^{t_b}, (\bullet z)^{t_c}, (\bullet xs)^{t_d} \rangle \\
 S_2 &= \langle H, z \mapsto 0^{t_e}, n1 \mapsto (\text{next } y \ z)^{t_4} \mid \text{foldl}^{t_s} \mid \\
 &\quad (\bullet \text{ next})^{t_3}, (\bullet n1)^{t_5}, (\bullet ys)^{t_6} \rangle \\
 S_3 &= \langle H, n1 \mapsto 1^{t_4}, n2 \mapsto (\text{next } y1 \ n1)^{t_4} \mid \text{foldl}^{t_s} \mid \\
 &\quad (\bullet \text{ next})^{t_3}, (\bullet n2)^{t_5}, (\bullet ys1)^{t_6} \rangle \\
 S_4 &= \langle H, n2 \mapsto 2^{t_4}, n3 \mapsto (\text{next } y2 \ n2)^{t_4} \mid \text{foldl}^{t_s} \mid \\
 &\quad (\bullet \text{ next})^{t_3}, (\bullet n3)^{t_5}, (\bullet ys2)^{t_6} \rangle
 \end{aligned}$$

Up until S_4 , supercompilation has proceeded untroubled by termination test failure. This changes when we reach S_4 , as the latest tag bag is identical to that for S_3 and so to ensure termination the supercompiler tries to *generalise* or *split* without reduction. Unfortunately, because we have two *equal* tag-bags, the supercompiler will be unable to identify a growing tag suitable for generalisation, and so will have to fall back on standard *split* rather than generalising away the n heap bindings as we would hope.

If we had not speculated, we would also not have the problem as $n1$ and $n2$ would remain unreduced. This would have turned the heap of S_3 to contain z — thus triggering the termination test at the point we supercompiled S_3 , and fingering t_4 as the growing tag. Furthermore, the problem would not occur if we had Peano numbers in place of literals and primops, because the growing nature of the accumulator would have shown up as an increasingly deeply nested application of S that is readily detectable by the growing-tags heuristic. Interestingly, as a side effect of the work-duplication check in the splitter, we also get good code “by chance” if we supercompile a strict `foldl`.

6.1 Our solution

Our solution to this problem is straightforward and non-invasive. The idea is that a binding such as $n2$ should “weigh” more heavily

Test	SC	Cmp.	Run	Mem.	Size
bernouilli	+3%	+0%	-2%	-1%	-2%
digitsofe2	+136%	+15%	-41%	-33%	+133%
exp3_8	+0%	+1%	+0%	+0%	-2%
primes	+2%	-2%	-1%	+0%	-30%
rfib	-2%	-2%	+0%	-1%	+0%
sumsquare	-16%	+0%	+10%	-80%	-28%
sumtree	+1%	+0%	+0%	-1%	+0%
x2n1	-32%	-2%	+4%	-84%	-45%
Average	+7%	+1%	-3%	-15%	+2%
Maximum	+136%	+15%	+10%	+0%	+133%
Minimum	-32%	-2%	-41%	-84%	-45%

Figure 8: The effect of enabling multiplicity tagging

in the mind of the growing-tags generaliser than a binding like $n1$ because its literal value embodies more computational history. To this end, we add a multiplicity to the type of tags, modeling a tag-bag with several repetitions of a single tag, vaguely approximating the effect of Peano arithmetic:

```
type Multiplicity = Int
type Tag = (Int, Multiplicity)
```

Tags are initialised with an *Multiplicity* of 1. All the tagged reduction rules of Figure 3 remain the same with the exception of PRIM-R, which is replaced with the following rule:

$$\langle H \mid v_2 \mid (v_1 \otimes \bullet)^{(x_0, c_0)}, K \rangle \rightsquigarrow \langle H \mid (\otimes(\ell_1, \ell_2))^{(x_0, \sum_j c_j)} \mid K \rangle$$

where $\ell_i^{(x_i, c_i)} = \text{deref}(H, v_i)$

This rule differs from the standard one in that it produces an output tag with an *Multiplicity* that has grown to be strictly greater than the tags of each of the inputs.

The final, crucial step is to make use of the *Multiplicity* information when constructing the *TagBag*:

$$\text{tagBag} \langle H \mid e^{(x, c)} \mid K \rangle = \{ \{ 2 * x \mid x \mapsto e^{(x, c)} \in H, 0 \leq i < c \} \}$$

$$\cup \{ \{ 3 * x \mid 0 \leq i < c \} \}$$

$$\cup \{ \{ 5 * x \mid \kappa^{(x, c)} \in K, 0 \leq i < c \} \}$$

No further changes are required — because we construct *TagBags* with several “copies” of a tag with an *Multiplicity* greater than one, the growing tags heuristic identifies them as growing in just the situations where we need it to. Consider our *length* function example above. In S_3 , the tag t_4 on $n1$ will have a multiplicity of 3, and that on $n2$ will be 1. By the time we reach S_4 the tag t_4 on $n2$ will have a count of 5 while the tag on $n3$ will be 1. Looking at the resulting tag bags, it is clear that t_4 is the growing tag and hence should serve as the basis of residualisation by the growing-tags heuristic.

6.2 Evaluating multiplicity tagging

In Figure 8 we show the effects of enabling our multiplicity tagging system for those programs that were affected by the change — as you might expect, only programs making use of primops and literals saw a change. One of the most dramatic results is for *sumsquare*, where the newly-discovered generalisation causes all of the lists allocated by the benchmark to be fused away, greatly improving memory usage and runtime. However, there are solid gains in runtime, memory usage and output size for all primop-using benchmarks — all originating from better generalisation.

7. Overall results

Our latest version of the Cambridge Haskell Supercompiler, CHSC2, incorporates all of the refinements described above: generalisation, rollback, speculation, and multiplicity tagging. We compare our

results to Supero 2010 [9] in Figure 9. The main difference is that CHSC2 discovers a better generalisation than Supero 2010 for *treeflip*, accounting for its ability to vastly reduce allocations there. It is not clear why CHSC’s versions of *digitsofe2* and *primes* fail to reduce memory allocation by as much as Supero. Stripping out the effect of *tak*, where Supero is unfairly penalised by the aforementioned strictness-analyser interaction, we obtain slightly better average memory consumption reduction (-41% vs -38%) and runtime reduction (-36% vs -30%).

Encouragingly, only *tak* and *exp3_8* got worse after supercompilation. Unfortunately, even after inspection of the supercompiled program it is unclear why the memory usage (and hence runtime) of *exp3_8* is made worse by supercompilation.

8. Related work

The tag-bag termination test was introduced by Mitchell [9]. The other major approach is to use a homeomorphic embedding on syntax trees [12] but supercompilers that use that method tend to spend most of their time evaluating the test [17].

Supero 2010 [9] used a generalisation mechanism for call-by-need supercompilers that use the tag-bag termination mechanism. The idea is that when computing a child term for recursive supercompilation, the splitting process avoids pushing down any bindings that would cause the new child state to immediately fail the termination check. We found that Supero’s generalisation method gives very similar results to our method in most situations. However, our method has two clear merits:

- It is easier to implement, because the splitter does not need to try inlining bindings in any particular order — the growing tags tell you exactly those bindings which may be inlined.
- It is faster: Supero’s method requires one termination test to be carried out for every binding you may wish to inline, whereas our method simply requires a set membership test per inlining.

The effect of rollback on the *sc* is present in standard supercompilers constructed around the idea of a “graph of configurations”, such as HOSC [18]. In such systems, when an earlier graph node a is embedded into a later node b , the a node (rather than the b node) is generalised. This has the effect of cutting off the part of the graph reachable via a , achieving rollback. We have described how this idea can be applied to a direct-style supercompiler, as well as how to apply the concept to “loopy” reduction and heap speculation.

Our decision to use speculative evaluation in the supercompiler was initially inspired by Ennals [19], though runtime speculation faces very different tradeoffs and design issues.

9. Conclusions and further work

We have described a highly-modular supercompiler whose performance is comparable to or better than prior work, with an average speedup and reduction in memory usage of 35%. We have shown how supercompilation can be improved by a number of local improvements to our earlier modular algorithm. By adding rollback to undo wasted supercompilation, we reduced both output program memory consumption and size by 16%. By adding a speculator for heap bindings, we improved information propagation and hence got more deforestation, reducing memory usage by another 14% on average. We also showed how the growing-tags heuristic could be used in a tag-bag based supercompiler to implement generalisation, reducing heap usage by 40%. Furthermore, we described how a supercompiler can make use of ubiquitously-normalised terms to help ensure that the output program never contains “obvious” reductions, and gave the first precise treatment of a tagged reduction semantics of the sort that is necessary for a tag-bag based supercompiler.

Test	Mitchell [9]					CHSC2				
	SC ^a	Cmp. ^b	Run ^c	Mem. ^d	Size ^e	SC ^a	Cmp. ^b	Run ^c	Mem. ^d	Size ^e
append	0.0s	+2%	-15%	-15%	+29%	0.0s	+0%	-11%	-12%	+170%
bernouilli	4.2s	+80%	-1%	-3%	+276%	0.1s	+5%	-5%	-6%	+162%
digitsofe2	3.1s	+14%	-56%	-53%	+15%	1.8s	+30%	-43%	-37%	+952%
exp3_8	0.6s	+18%	+1%	+0%	+559%	0.0s	+2%	+6%	+11%	+119%
factorial	0.0s	-3%	+1%	+0%	-23%	0.0s	-1%	-1%	+0%	+19%
primes	0.0s	+2%	-38%	-23%	-21%	0.0s	-1%	-40%	-9%	+9%
raytracer	0.0s	+1%	-42%	-56%	+54%	0.0s	-2%	-45%	-55%	+64%
rfib	0.0s	+1%	-11%	+0%	-13%	0.0s	+0%	-28%	-1%	+125%
sumsquare	13.8s	+28%	-62%	-100%	+638%	0.0s	+0%	-98%	-100%	+71%
sumtree	0.0s	+0%	-83%	-100%	+50%	0.0s	+0%	-90%	-100%	+60%
tak	0.0s	+3%	-38%	+23018%	-41%	0.0s	+2%	-15%	+37%	+229%
treeflip	0.0s	+1%	-44%	-45%	+99%	0.0s	+0%	-90%	-100%	+52%
x2n1	0.0s	+4%	-13%	-66%	+39%	0.0s	+0%	+5%	-84%	+35%
Average		+12%	-31%	+1735%	+128%		+3%	-35%	-35%	+159%
Maximum		+80%	+1%	+23018%	+638%		+30%	+6%	+37%	+952%
Minimum		-3%	-83%	-100%	-41%		-2%	-98%	-100%	+9%

- ^a Supercompilation time
^b GHC compile time vs. GHC - O2
^c Program runtime vs. GHC - O2
^d Runtime allocation vs. GHC - O2
^e Size increase (in AST nodes) due to supercompilation

Figure 9: Overall benchmark results

We have not yet proven that our modified supercompiler terminates, though we have taken care to avoid sources of non-termination in every component of our design. The supercompiler does terminate on all examples we have tried. Unfortunately, it does not seem possible to directly reuse our previous termination from [7] because of our use of term normalisation, so an updated (and elaborated) termination proof is an obvious line of further work.

An interesting direction would be to compare our generalisation heuristic with the MSG, and do a more complete comparison with Supero 2010's generalisation.

We hope to incorporate a supercompiler into GHC itself so we can experiment with supercompiling larger, real world programs. To supercompile arbitrary programs, we are exploring means to control the explosive growth in the number of distinct specialisations that occurs for many common examples.

Acknowledgments

This work was partly supported by a PhD studentship generously provided by Microsoft Research.

References

- [1] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.
- [2] Alexei P. Lisitsa and Andrei P. Nemytykh. Verification as specialization of interpreters with respect to data. In *Proceedings of First International Workshop on Metacomputation*, pages 94–112, 2008.
- [3] 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.
- [4] 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.
- [5] Simon Peyton Jones. Constructor specialisation for Haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 327–337, 2007.
- [6] 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.
- [7] Max Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, September 2010.
- [8] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03):231–264, 1997.
- [9] Neil Mitchell. Rethinking supercompilation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*. ACM, 2010.
- [10] Valentin F. Turchin. The algorithm of generalization in the supercompiler. *Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, Partial Evaluation and Mixed Computation*, pages 531–549, 1988.
- [11] Ilya Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010.
- [12] 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.
- [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] Jan Kort. Deforestation of a raytracer. *Master's thesis, Department of Computer Science, University of Amsterdam, The Netherlands*, 1996.
- [15] 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.
- [16] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.
- [17] 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.
- [18] Ilya Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.
- [19] Robert Ennals and Simon Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, New York, NY, USA, 2003. ACM.