Taming Code Explosion in Supercompilation

Peter A. Jonsson   Johan Nordlander

Luleå University of Technology
{pj, nordland}@csee.ltu.se

Abstract

Supercompilation algorithms can perform great optimizations but sometimes suffer from the problem of code explosion. This results in huge binaries which might hurt the performance on a modern processor. We present a supercompilation algorithm that is fast enough to speculatively supercompile expressions and discard the result if it turned out bad. This allows us to supercompile large parts of the imaginary and spectral parts of nofib in a matter of seconds while keeping the binary size increase below 5%.

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors – Compilers, Optimization;   D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages

General Terms   Algorithms, Performance

Keywords   supercompilation, deforestation, Haskell

1. Introduction

Supercompilation is an automatic program transformation that subsumes several program transformations which are available in production compilers [29]. It is well known that the transformation is correct [22] and terminates [25]. We also know that supercompilation gives remarkable optimization results on small examples: a textbook example of word counting in Haskell performs better than a low level implementation written in C [16].

Given these accomplishments, why does not your favorite compiler include a supercompiler? The two main problems are scalability of the supercompiler and the risk for code explosion. Existing implementations of supercompilers trade compilation time against great optimization and do not take the resulting program size into account. Closer investigation reveals that it is the test for non-termination, the homeomorphic embedding, that takes up to 50% of the transformation time [16]. We suggest sacrificing some optimization in exchange for faster compilation by revising the design of the supercompilation algorithm.

The faster algorithm allows us to control code explosion by speculatively supercompile expressions and discard bad results. Even incredibly cheap and simple tests can get a long way towards the goal: a surprising discovery is that comparing the number of syntactic nodes of the resulting expression with a well-chosen constant is a remarkably accurate predictor of "bad optimizations" that

should be discarded. Except this size threshold our measurements use a simple comparison of the size of the resulting expression versus the number of reduction steps performed by the supercompiler.

The performance of our algorithm rests on two important design choices:

Term Representation: We represent terms as zippers [4] during the transformation, which gives efficient and convenient implementations of tests for non-termination and splitting of terms. As a side effect this representation also makes it easy to reduce the number of terms to test for non-termination: only test against those terms who have the same head as the current term.

Term Size in Focus: By decreasing the size of the term that our supercompiler focuses on we can effectively decrease the cost of testing for non-termination. This shows in our compilation times: they are less than 3 seconds on a modern desktop computer for all the small examples from the nofib suite [18].

Our design allows us to supercompile most programs from the spectral part of nofib in less than three seconds. Some of these programs are 10 times larger than previously supercompiled Haskell programs. We do not believe that an ordinary programmer will turn on the supercompiler for his development builds, but it is certainly reasonable to do so for testing and nightly builds.

Our contributions are:

- A mechanism that is parametric with respect to a metric that allows us to control code explosion (Section 3.2 and Section 4.6).

- A practical supercompilation algorithm (Section 4)

- Measurements on the imaginary and spectral parts of nofib (Section 5).

- Implementation tricks to speed up the actual implementation (Section 6).

2. Language

Our language of study is a higher-order functional language with let-bindings, case-expressions and primitive values. Its syntax for expressions, values and patterns is shown in Figure 1.

Let F range over a set $\mathcal{F}$ of global function definitions, where all functions have a specific arity and contain no free variables. This is not a severe restriction on the programmer since the input can be lambda lifted [5].

The language contains integer values n and arithmetic operations $\oplus$, although these meta-variables can preferably be understood as ranging over primitive values in general and arbitrary operations on these. We let $+$ denote the semantic meaning of $\oplus$.

A list of expressions $e_1 \dots e_n$ is abbreviated as $\overline{e}$, and a list of variables $x_1 \dots x_n$ as $\overline{x}$.

We denote the free variables of an expression e by $fv(e)$ and a program is an expression with no free variables and all function

Expressions

$$e \quad ::= \quad n \mid x \mid F \mid K \mid e\,e' \mid \lambda x.e \mid e \oplus e'$$
$$\quad\quad\quad | \quad \texttt{case } e \texttt{ of } \{p_i \rightarrow e_i\} \mid \texttt{let } x = e \texttt{ in } e'$$

$$p \quad ::= \quad n \mid K\,\overline{x}$$

Weak Head Normal Form

$$w \quad ::= \quad n \mid \lambda x.e \mid K\,\overline{e}$$

Figure 1. The language

Reduction contexts

$$\mathcal{E} \quad ::= \quad \square \mid \mathcal{E}\,e \mid \mathcal{E} \oplus e \mid n \oplus \mathcal{E} \mid \texttt{case } \mathcal{E} \texttt{ of } \{p_i \rightarrow e_i\}$$

Evaluation relation

$[F]\mathcal{E}$	$\mapsto$	$[e]\mathcal{E}$, if $F \stackrel{\text{def}}{=} e$	(Global)
$[(\lambda x.e)\,e']\mathcal{E}$	$\mapsto$	$[[e'/x]e]\mathcal{E}$	(Beta)
$[\texttt{let } x = e \texttt{ in } e']\mathcal{E}$	$\mapsto$	$[[e/x]e']\mathcal{E}$	(Let)
$[\texttt{case } K\,\overline{e} \texttt{ of } \{K_i\,\overline{x}_i \rightarrow e_i\}]\mathcal{E}$	$\mapsto$	$[[\overline{e}/\overline{x}_j]e_j]\mathcal{E}$	(KCase)
		if $K = K_j$	
$[\texttt{case } n \texttt{ of } \{n_i \rightarrow e_i\}]\mathcal{E}$	$\mapsto$	$[e_j]\mathcal{E}$, if $n = n_j$	(NCase)
$[n_1 \oplus n_2]\mathcal{E}$	$\mapsto$	$[n]\mathcal{E}$, if $n = n_1 + n_2$	(Arith)

Figure 2. Reduction semantics

names defined in $\mathcal{F}$. The operational semantics is call-by-name as defined in Figure 2. Capture free substitution expressions $\overline{e}$ for variables $\overline{x}$ in e' is denoted by $[\overline{e}/\overline{x}]e'$.

A reduction context $\mathcal{E}$ is a term containing a single hole $\square$, which indicates the next expression to be reduced. The expression $[e]\mathcal{E}$ is the term obtained by replacing the hole in $\mathcal{E}$ with e.

3. Design Choices

This section describes a number of important design choices and how they contribute to our goal of making a practical supercompiler for program optimization.

Our guiding principle has been simple: sacrifice optimization opportunities if it significantly improves the performance or simplifies the implementation of the supercompiler. Our only concern is soundness of the supercompiler, we do not have to worry about completeness properties that could be important in theorem proving or similar domains. We therefore design a positive supercompiler, but these choices should be straightforward to transfer to other more powerful supercompilers.

There is some overlap between what problem each design choice helps to protect against.

Code Explosion The primary mechanism to tame code explosion is described in Section 3.2. Section 3.1, Section 3.3, and Section 3.4 contain design choices that help reduce the code size.

Compiler Performance Avoiding code explosion helps compilation performance, but this is not enough: Section 3.5 contains a mechanism to avoid performing work that is unlikely to give any benefit.

We have also made a lot of minor non-controversial design choices in our supercompiler which we describe in Section 3.6.

3.1 Boring Contexts

There are sometimes function calls that have no static information in their argument, making it impossible to improve them by super-

compiling them in their context. By using a simple approximation of such function calls we save compilation time and avoid creating specializations that do not improve the performance. We say that such functions are in a boring context. A boring expression is an expression of the form:

$$b ::= x \mid b\,b \mid n \oplus b \mid b \oplus n \mid b \oplus b$$

A context $\mathcal{R}$ is defined as boring iff $\mathcal{R} = \epsilon$ or $\mathcal{R} = \square\,b : \mathcal{R}'$ and $\mathcal{R}'$ is boring.

3.2 Discarding Expressions

The initial function call to reverse with an accumulating parameter, *reverse xs* $[]$, is an example that will pass the test for boring expressions since the second argument is a known constructor. Despite this there is not much the supercompiler can do with this expression and the end result will be a new function with one unfolding of reverse and a second function that is isomorphic to reverse.

Designing an algorithm that approximates what the result and how much savings were achieved from supercompiling an arbitrary expression is difficult. We can completely avoid to solve this problem by speculatively supercompile the expression, have the supercompiler keep track of the reduction steps taken, and look at the output of the supercompiler. If the number of reduction steps taken is small or the code size is deemed too large we discard the supercompiled expression and supercompile the subexpressions of the current expression instead.

3.3 Normalization of Expressions

We perform a simple normalization of expressions, as specified in Section 4.5, before comparing them for folding possibilities. This makes the supercompiler significantly faster. The big win is the removal of identity coercions in expressions, which makes the memoization list a lot shorter.

3.4 Sharing New Function Definitions

It is quite common for the case of case-rule to fire. A typical case looks something like this:

case (**case** x **of**
$\quad\quad p_1 \ \rightarrow \ x_1$
$\quad\quad \cdots$
$\quad\quad p_n \ \rightarrow \ x_n$) **of**
$[] \ \rightarrow \ []$
$(x' : xs') \ \rightarrow \ append\ xs'\ zs$

Once the case of case-rule is done the code is:

case x **of**
$\quad p_1 \ \rightarrow \ $ **case** x_1 **of**
$\quad\quad\quad\quad [] \ \rightarrow \ []$
$\quad\quad\quad\quad (x' : xs') \ \rightarrow \ append\ xs'\ zs$
$\quad \cdots$
$\quad p_n \ \rightarrow \ $ **case** x_n **of**
$\quad\quad\quad\quad [] \ \rightarrow \ []$
$\quad\quad\quad\quad (x' : xs') \ \rightarrow \ append\ xs'\ zs$

The driving algorithm will transform the call to append twice, and create two functions that are both isomorphic to append. Creating the functions at the top level and calling the same function from both branches saves both transformation work and reduces code size.

Using a state monad for the driving algorithm, and have that monad store a memoization list augmented with the function definitions will allow the second branch to just insert a call to this new function created in the first branch, thereby saving a lot of transformation effort.

However, if we fold too eagerly this will miss out on specialization opportunities and this will make the final program perform

worse than necessary. An example is $append\ (append\ xs\ ys)\ zs$, which when transformed will create a function isomorphic to append itself in the first branch. After the supercompiler has transformed the second branch it will find a recursive call to $append\ (append\ xs'\ ys)\ zs$, which can fold against both the call in the first branch, $append\ ys\ zs$, and the initial call that the transformation started from. If the former is chosen no fusion will occur.

We therefore only look for renamings, not instances, when trying to fold against things stored in the monad. If there exists a specialised version of our current function the algorithm will use that, and otherwise it will create a new specialised copy. Our algorithm will only generalize expressions against their ancestors, never across branches of case-expressions.

3.5 Tainting Functions

If the same function, regardless of context, is supercompiled in different parts of the program and the result is discarded more than a certain number of times we make the unfolding for the function unavailable. This prevents future attempts of specializing the function. The concrete mechanism is that we add a special triple on the form $(*, \emptyset, F)$ to the store each time we discard a result from supercompiling a function F in some context, and the number of such triples in the store is checked for before inlining a function.

One might ask whether innocent functions will be tainted when trying to supercompile complex expressions, for example $id\ e$ where e is an expression that will be discarded after supercompilation, which will prevent other callsites from benefiting of supercompilation. Since the supercompiled e is discarded the supercompiler will compare e and $id\ e$ with the extra information that the transformation has saved one beta-reduction. Whether this is a worthwhile saving is a question of balancing the acceptable-predicate.

We used being discarded 3 times as threshold for our measurements.

3.6 Miscellaneous Design Choices

When supercompiling a function call $const\ x\ y$ our supercompiler blindly emits a call to a fresh function $h\ x\ y$ whose body is defined as $(\lambda x\ y.x)\ x\ y$. It is safe to directly inline the body of $const$ at the call-site but we can increase the code sharing by allowing the supercompiler to fold against this definition from other parts of the program.

If it turns out that there is only one call to h in the entire supercompiled program the ordinary simplifier of the compiler will inline it.

The reason we use evaluation contexts in our algorithm is that there is a chance that good things can happen in the interaction between evaluation contexts and an expression in the hole. There are no such possibilities with an arbitrary context that has the hole outside the focus for evaluation.

4. Supercompilation

This section contains the definition of our supercompilation algorithm and its dependencies.

4.1 Preliminaries

We replace our nested evaluation context $\mathcal{E}$ with a list of shallow reduction contexts for our supercompiler as defined in Figure 3. It will turn out to be convenient for the implementation to pattern match on the context if it is a list of single level contexts. We let $\overline{\Box\ e}$ denote a list of application contexts.

If a variable appears no more than once in an expression, that expression is said to be *linear* with respect to that variable. Like Wadler [31], we extend the definition slightly for linear case ex-

The outer context:
$$\mathcal{R} ::= \epsilon \mid \mathcal{G} : \mathcal{R}$$

Each frame:
$$\mathcal{G} ::= \Box\ e \mid \Box \oplus e \mid n' \oplus \Box \mid \texttt{case}\ \Box\ \texttt{of}\ \{p_i \rightarrow e_i\}$$

Memoization element:
$$\nu ::= (h, \overline{x}, [F]\mathcal{R})$$

Memoization list and store:
$$\begin{aligned} \rho &::= \epsilon \mid \nu : \rho \\ \sigma &::= \epsilon \mid (\nu, e') : \sigma \end{aligned}$$

Figure 3. Context definition

pressions: no variable may appear in both the selector and a branch, although a variable may appear in more than one branch.

4.2 Algorithm Definition

Our supercompiler is defined as two mutually recursive functions that pattern-match on expressions and contexts in order to rewrite expressions. The first algorithm is called the *driving* algorithm, used to perform evaluation steps, and the second algorithm is called the *building* algorithm, which reassembles the transformed expression. These algorithms are defined in Figure 4. Both algorithms take five parameters: 1) an expression; 2) a memoization list ρ; 3) the context $\mathcal{R}$; 4) the store σ; and 5) the savings from the supercompiler so far. New function definitions are put in the store. The memoization list holds information about expressions already traversed and is used to guarantee termination. A notational convention is to use e' to denote expressions that are already transformed.

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R11-R13 are the default focusing rules that extend the given driving context $\mathcal{R}$ and zoom in on the next expression to be transformed. If no other rule matches, such as for a single variable, rule R14 will match and call build to rebuild the expression.

The program is turned "inside-out" by moving the surrounding context $\mathcal{R}$ into all branches of the case-expression through rules R19 and R20. Rule R10 has a similar mechanism for let-expressions. Rule R9 is only allowed to match if the variable y is not freshly generated by the splitting mechanism described in the Section 4.4. This side condition ensures that the expression splitting and reassembly performed by $\mathcal{D}_{app}(\ )$ will not cause arbitrary code and work duplication.

Rules R3 and R4 refer to $\mathcal{D}_{app}(\ )$ and $\mathcal{D}_{varapp}(\ )$, both defined in Figure 5. They can both be inlined in the definition of the driving algorithm, they are merely given a separate name for improved clarity of the presentation.

The fifth parameter to the supercompiler, the savings, is there to track how many reduction steps the supercompiler has performed. This is currently represented as a tree in our implementation to aid us in the design of when to discard expressions, but a production compiler should use a more efficient representation. We assign the value 1 to ordinary evaluation steps in rules R1, R2, R6, R8, R9 and R10. Rule R5, the case reduction with a known constructor, is assigned the value 5 since it both decreases code size and removes a memory allocation.

We need some more arithmetic operations on our savings beyond plain addition: # indicates that a saving was under a lambda, ? combines branches of savings such as in rule R10 and max of all savings in rules R19 and R20. The most important saving, folding, is denoted by the exclamation mark (!). These abstract savings need to be interpreted to a concrete number in order to make deci-

General form:
$$\mathcal{D}\llbracket e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s = (e', \sigma', s')$$

Evaluation Rules

$\mathcal{D}\llbracket n_j \rrbracket \, \rho \, (\texttt{case} \; \Box \; \texttt{of} \; \{n_i \to e_i\} : \mathcal{R}) \, \sigma \, s$	$=$	$\mathcal{D}\llbracket e_j \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+1)$	(R1)
$\mathcal{D}\llbracket n \rrbracket \, \rho \, (n_1 \oplus \Box : \mathcal{R}) \, \sigma \, s$	$=$	$\mathcal{D}\llbracket n_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+1), \text{where } n_2 = n_1 + n$	(R2)

$\mathcal{D}\llbracket F \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s \mid F \text{ not tainted and } \mathcal{R} \text{ not boring}$	$=$	$\mathcal{D}_{app}(F) \, \rho \, \mathcal{R} \, \sigma \, s$	(R3)

$\mathcal{D}\llbracket x \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\mathcal{D}_{varapp}(x) \, \rho \, \mathcal{R} \, \sigma \, s$	(R4)

$\mathcal{D}\llbracket K_j \rrbracket \, \rho \, (\overline{\Box e} : \texttt{case} \; \Box \; \texttt{of} \; \{K_i \, \overline{x}_i \to e_i\} : \mathcal{R}) \, \sigma \, s$	$=$	$\mathcal{D}\llbracket \texttt{let} \, \overline{x}_j = \overline{e} \, \texttt{in} \, e_j \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+5)$	(R5)

$\mathcal{D}\llbracket \lambda \overline{x}.e_1 \rrbracket \, \rho \, (\overline{\Box \; e_2} : \mathcal{R}) \, \sigma \, s$	$=$	$\mathcal{D}\llbracket \texttt{let} \, \overline{x} = \overline{e_2} \, \texttt{in} \, e_1 \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+1)$	(R6)
$\mathcal{D}\llbracket \lambda x.e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\textbf{let} \, (e', \sigma_1, s') = \mathcal{D}\llbracket e \rrbracket \, \rho \, \epsilon \, \sigma \, 0$	(R7)
		$\textbf{in} \, \mathcal{B}\llbracket \lambda x.e' \rrbracket \, \rho \, \mathcal{R} \, \sigma_1 \, (s + \#s')$	

$\mathcal{D}\llbracket \texttt{let} \, x = n \, \texttt{in} \, e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\mathcal{D}\llbracket [n/x]e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+1)$	(R8)
$\mathcal{D}\llbracket \texttt{let} \, x = y \, \texttt{in} \, e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s \mid y \text{ not freshly generated}$	$=$	$\mathcal{D}\llbracket [y/x]e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+1)$	(R9)
$\mathcal{D}\llbracket \texttt{let} \, x = e_1 \, \texttt{in} \, e_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s \mid \text{linear } x \, e_2$	$=$	$\mathcal{D}\llbracket [e_1/x]e_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma \, (s+1)$	(R10)
$\mid \text{otherwise}$	$=$	$\textbf{let} \, (e'_1, \sigma_1, s_1) = \mathcal{D}\llbracket e_1 \rrbracket \, \rho \, \epsilon \, \sigma \, 0$	
		$\quad (e'_2, \sigma_2, s_2) = \mathcal{D}\llbracket e_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma_1 \, s$	
		$\textbf{in} \, (\texttt{let} \, x = e'_1 \, \texttt{in} \, e'_2, \sigma_2, ?s_1 + s_2)$	

Focusing Rules

$\mathcal{D}\llbracket e_1 \oplus e_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\mathcal{D}\llbracket e_1 \rrbracket \, \rho \, (\Box \; \oplus e_2 : \mathcal{R}) \, \sigma \, s$	(R11)
$\mathcal{D}\llbracket e_1 \, e_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\mathcal{D}\llbracket e_1 \rrbracket \, \rho \, (\Box \; e_2 : \mathcal{R}) \, \sigma \, s$	(R12)
$\mathcal{D}\llbracket \texttt{case} \, e \, \texttt{of} \, \{p_i \to e_i\} \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\mathcal{D}\llbracket e \rrbracket \, \rho \, (\texttt{case} \; \Box \; \texttt{of} \; \{p_i \to e_i\} : \mathcal{R}) \, \sigma \, s$	(R13)

Fallthrough

$\mathcal{D}\llbracket e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	$=$	$\mathcal{B}\llbracket e \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	(R14)

Rebuilding Expressions

$\mathcal{B}\llbracket n \rrbracket \, \rho \, (\Box \; \oplus e_2 : \mathcal{R}) \, \sigma \, s$	$=$	$\mathcal{D}\llbracket e_2 \rrbracket \, \rho \, (n \oplus \Box : \mathcal{R}) \, \sigma \, s$	(R15)
$\mathcal{B}\llbracket e' \rrbracket \, \rho \, (\Box \; \oplus e_2 : \mathcal{R}) \, \sigma \, s$	$=$	$\textbf{let} \, (e'_2, \sigma_1, s') = \mathcal{D}\llbracket e_2 \rrbracket \, \rho \, \epsilon \, \sigma \, s$	(R16)
		$\textbf{in} \, \mathcal{B}\llbracket e' \oplus e'_2 \rrbracket \, \rho \, \mathcal{R} \, \sigma_1 \, s'$	
$\mathcal{B}\llbracket e' \rrbracket \, \rho \, (e'_1 \oplus \Box : \mathcal{R}) \, \sigma \, s$	$=$	$\mathcal{B}\llbracket e'_1 \oplus e' \rrbracket \, \rho \, \mathcal{R} \, \sigma \, s$	(R17)
$\mathcal{B}\llbracket e' \rrbracket \, \rho \, (\Box \, e : \mathcal{R}) \, \sigma \, s$	$=$	$\textbf{let} \, (e'', \sigma_1, s') = \mathcal{D}\llbracket e \rrbracket \, \rho \, \epsilon \, \sigma \, s$	(R18)
		$\textbf{in} \, \mathcal{B}\llbracket e' e'' \rrbracket \, \rho \, \mathcal{R} \, \sigma_1 \, s'$	
$\mathcal{B}\llbracket x' \rrbracket \, \rho \, (\texttt{case} \; \Box \; \texttt{of} \; \{p_i \to e_i\} : \mathcal{R}) \, \sigma \, s$	$=$	$\textbf{let} \, (e'_i, \sigma_i, s_i) = \mathcal{D}\llbracket [p_i/x]e_i \rrbracket \, \rho \, ([p_i/x]\mathcal{R}) \, \sigma_{i-1} \, s$	(R19)
		$\textbf{in} \, (\texttt{case} \, x' \, \texttt{of} \, \{p_i \to e'_i\}, \sigma_i, max(s_i))$	
$\mathcal{B}\llbracket e' \rrbracket \, \rho \, (\texttt{case} \; \Box \; \texttt{of} \; \{p_i \to e_i\} : \mathcal{R}) \, \sigma \, s$	$=$	$\textbf{let} \, (e'_i, \sigma_i, s_i) = \mathcal{D}\llbracket e_i \rrbracket \, \rho \, \mathcal{R} \, \sigma_{i-1} \, s$	(R20)
		$\textbf{in} \, (\texttt{case} \, e' \, \texttt{of} \, \{p_i \to e'_i\}, \sigma_i, max(s_i))$	
$\mathcal{B}\llbracket e' \rrbracket \, \rho \, \epsilon \, \sigma \, s$	$=$	(e', σ, s)	(R21)

Figure 4. Driving algorithm and building algorithm

sions on them. For our measurements we used regular addition of savings both under lambdas and in branches such as rule R10, and gave folding an extremely high constant value (500). Our design principle was that foldings should trump all other kinds of savings, but this is an extremely coarse design that leaves plenty of room for future improvements.

We use the operator $\rho|_F$ to extract all expressions with the head F from ρ. Our implementation stores the partitioned ρ's to avoid repeating the partitioning. Partitioning the memoization list gives significant time savings, but it can only be applied on the outermost level.

$\mathcal{D}_{varapp}()$ will try to fold against expressions with a variable in the head. This is convenient and reduces code size in general. A good example is supercompiling the right hand sides of append in a module:

$$append \; xs \; ys \; = \mathcal{D}\llbracket \; \texttt{case} \; xs \; \texttt{of} \; \{\dots\} \; \rrbracket$$
$\{ \textit{Create a fresh function and transform its body} \}$

$$\begin{aligned}
append \; xs \; ys \; &= \; h \; xs \; ys \\
h \; xs \; ys \; &= \; \textbf{case} \; xs \; \textbf{of} \\
& \qquad [] \; \to \; ys \\
& \qquad (x' : xs') \; \to \; x' : \mathcal{D}\llbracket \; append \; xs' \; ys \rrbracket
\end{aligned}$$
$\{ \textit{Inline append} \}$
$$\begin{aligned}
append \; xs \; ys \; &= \; h \; xs \; ys \\
h \; xs \; ys \; &= \; \textbf{case} \; xs \; \textbf{of} \\
& \qquad [] \; \to \; ys \\
& \qquad (x' : xs') \; \to \; x' : \mathcal{D}\llbracket \; \texttt{case} \; xs' \; \texttt{of} \; \{\dots\} \rrbracket
\end{aligned}$$
$\{ \textit{Fold against h} \}$
$$\begin{aligned}
append \; xs \; ys \; &= \; h \; xs \; ys \\
h \; xs \; ys \; &= \; \textbf{case} \; xs \; \textbf{of} \\
& \qquad [] \; \to \; ys \\
& \qquad (x' : xs') \; \to \; x' : h \; xs' \; ys
\end{aligned}$$

Without variable folding this example would give one more unrolling of the recursion which would have increased the code size.

$$\mathcal{D}_{app}(F)\,\rho\,\mathcal{R}\,\sigma\,s\,|\exists(h,\overline{y},e_1)\in(\sigma|_F\cup\rho|_F)\,.\,[\overline{x}/\overline{y}]e_1=e_2 \quad=\quad (h\,\overline{x},\sigma,!s) \tag{1}$$
$$|\exists(h,\overline{y},e_i)\in(\rho|_F)\,.\,e_1\trianglelefteq^* e_2 \quad=\quad \textbf{let}\,(f_g,[\overline{f}/\overline{y}])=divide(e_2,e_1) \tag{2}$$
$$(\overline{f}',\sigma_i,s_i)=\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon\,\sigma_{i-1}\,s_{i-1}$$
$$(f_g',\sigma_2,s_2)=\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon\,\sigma_i\,s_i$$
$$\textbf{in}\,([\overline{f}'/\overline{y}]f_g',\sigma_2,max(s_i,s_2))$$
$$|\,acceptable(e_2,e',s_1) \quad=\quad (h\,\overline{x},((h,\overline{x},e_2),e'):\sigma_1,s_1) \tag{3}$$
$$|\,\text{otherwise} \quad=\quad \mathcal{B}[\![F]\!]\,\rho\,\mathcal{R}\,((*,\emptyset,F):\sigma)\,s \tag{4}$$
$$\text{where}\ (e',\sigma_1,s_1)=\mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R}\,\sigma s$$
$$F\stackrel{def}{=}e,\ \rho'=(h,\overline{x},e_2):\rho,\ \overline{x}=fv(e_2),\ h\ \text{fresh},\ e_2=normalize([F]\mathcal{R})$$

$$\mathcal{D}_{varapp}(x)\,\rho\,\mathcal{R}\,\sigma\,s\,|\exists(h,\overline{y},e_1)\in(\sigma\cup\rho)\,.\,[\overline{x}/\overline{y}]e_1=e_2 \quad=\quad (h\,\overline{x},\sigma,!s) \tag{5}$$
$$|\,acceptable(e_2,e',s_1) \quad=\quad (h\,\overline{x},((h,\overline{x},e_2),e'):\sigma_1,s_1) \tag{6}$$
$$|\,\text{otherwise} \quad=\quad \mathcal{B}[\![x]\!]\,\rho\,\mathcal{R}\,\sigma\,s \tag{7}$$
$$\text{where}\ (e',\sigma_1,s_1)=\mathcal{B}[\![x]\!]\,\rho'\,\mathcal{R}\,\sigma\,s$$
$$e_2=normalize([x]\mathcal{R}),\ \rho'=(h,\overline{x},e_2):\rho,\ \overline{x}=fv(e_2),\ h\ \text{fresh}$$

Figure 5. Driving of applications

4.3 Termination Checking

Folding by itself is not sufficient to ensure termination of the driving algorithm. We therefore define a predicate which is commonly referred to as "the whistle" in literature on supercompilation. Whenever the whistle blows, our algorithm splits the input expression into strictly smaller expressions that are transformed separately in the empty context. This might expose new folding opportunities, and allows the algorithm to remove intermediate structures in subexpressions.

The design of the predicate is inspired by the homeomorphic embedding relation [26] but adjusted to work on our zipper representation and to admit more expressions. A specific design criterion was that $append\ xs\ ys$ must not be embedded in $append\ xs\ xs$. It is easy to construct infinite chains of expressions that will be admitted by this predicate by itself, $append\ xs\ ys$, $append\ xs\ ys$, ... is one example. However, non-termination is still avoided because the folding mechanism will prevent such chains to build up in ρ.

Definition 4.1 ($\trianglelefteq^*$)**.**

$$\frac{sizeof([e_1]\mathcal{R}) < sizeof([e_2]\mathcal{R}')}{[e_1]\mathcal{R} \trianglelefteq^* [e_2]\mathcal{R}' = [e_1]\mathcal{R} \trianglelefteq [e_2]\mathcal{R}'}$$

Definition 4.2 ($\trianglelefteq$)**.**

$$x \trianglelefteq y,\quad n_1 \trianglelefteq n_2,\quad \frac{e_1,e_2 \neq \lambda x.e,\ e_1 = e_2,\ \mathcal{R} \trianglelefteq \mathcal{R}'}{[e_1]\mathcal{R} \trianglelefteq [e_2]\mathcal{R}'}$$

$$\epsilon \trianglelefteq \epsilon,\quad \frac{\mathcal{R} \trianglelefteq \mathcal{R}'\ \mathcal{G} \trianglelefteq \mathcal{G}'}{\mathcal{G}:\mathcal{R} \trianglelefteq \mathcal{G}':\mathcal{R}'},\quad \frac{\mathcal{R} \trianglelefteq \mathcal{R}'}{\mathcal{R} \trianglelefteq \mathcal{G}':\mathcal{R}'}$$

$$\frac{e \trianglelefteq e'}{\Box\,e \trianglelefteq \Box\,e'},\quad \frac{e \trianglelefteq e'}{\Box \oplus e \trianglelefteq \Box \oplus e'},\quad \frac{e \trianglelefteq e'}{e \oplus \Box \trianglelefteq e' \oplus \Box},$$

$$\frac{e_i \trianglelefteq e_i'}{\texttt{case}\ \Box\ \texttt{of}\{p_i \to e_i\} \trianglelefteq \texttt{case}\ \Box\ \texttt{of}\{p_i \to e_i'\}},$$

We postpone dealing with the problem of non-termination of the supercompiler in the case of recursive data types that appear contravariantly in their own definition. As an aside, we note that the inliner in GHC will also loop on such programs [19].

4.4 Expression Splitting

When the whistle has blown it is necessary to stop transforming the current expression. It is safe to just leave it in place, but by splitting the expression and transforming the parts separately it is possible to achieve more optimization.

Divide tries to preserve as much structure between its first input argument and output by first calling an adaptation of the most specific generalisation (msg) [26] to the zipper representation. Our adaption is a bit unconventional in that it returns an expression and a substitution rather than the standard of an expression and two substitutions. Should the call to msg fail, *divide* instead calls split which will always return a valid result under the conditions ensured by the driving algorithm: *divide* is always called on two expressions on the form $[F]\mathcal{R}$ and $[F]\mathcal{R}'$. Divide, split and msg are defined in Figure 6. For the presentation we ignore the issues concerned with splitting under binders in expressions since binders are treated in the same way as described by Pfenning [20].

The msg can return an expression which is of greater size than its input because of binders: when splitting the two expressions $\texttt{case}\ x\ \texttt{of}\ \{\ (a,\ b)\ \to\ a\ \}$ and $\texttt{case}\ x\ \texttt{of}\ \{\ (a,\ b)\ \to\ b\ \}$ the result will be $(\texttt{case}\ x\ \texttt{of}\ \{\ (a,\ b) \to\ z\ a\ \},\ [(\lambda\ a.a)/z])$.

4.5 Normalization of Expressions

Finding renamings of an expression is a syntactical check, and sometimes this will miss out opportunities to fold because expressions that are semantically equivalent have different forms.

Consider the two expressions $F\ x\ (1\ +\ 2)$ and $F\ y\ (2\ +\ 1)$ – it is obvious to a human that these two expressions should be foldable against each other. By performing some kind of limited normalization on these expressions the second parameter will turn out to be 3 in both cases and they can fold. We can not hope to find a normal form for our expressions in general, but performing ordinary evaluation except inlining of function calls on all subexpressions will get a long way towards our goal. It is of course important to have a normalization that terminates.

A word of caution about making the normalization too powerful: many common optimizations change the structure of expressions. One example is changing the expression $(x + 1) + 1$ to $x + 2$ which is semantically correct and clearly saves computations, but chances are that in practice the first expression came from an expression $y + 1$ in a recursive call with an increasing parameter. Both $(y + 1)$ will be homeomorphically embedded in both $(x\ +\ 1)\ +\ 1$ and $x\ +\ 2$, but the former will be generalized to $(z\ +\ 1,\ [(x+1)/z])$ and which in turn will fold nicely against $y\ +\ 1$. The normalization scheme we use is the following rewrite rules applied exhaustively on all parts of expressions:

$$n_1 \oplus n_2 \quad\leadsto\quad n,\ \text{where}\ n = n_1 + n_2$$
$$\texttt{case}\ K_j\ \overline{e}\ \texttt{of}\ \{p_i \to e_i\} \quad\leadsto\quad \texttt{let}\ \overline{x_j} = \overline{e}\ \texttt{in}\ e_j$$
$$\texttt{case}\ n_j\ \texttt{of}\ \{p_i \to e_i\} \quad\leadsto\quad e_j$$
$$\texttt{let}\ x = n\ \texttt{in}\ e \quad\leadsto\quad [n/x]e$$

$$divide([e_1]\mathcal{R}, [e_2]\mathcal{R}') = (e, \theta) \qquad\qquad\qquad\qquad \text{iff } (e, \theta) = msg([e_1]\mathcal{R}, [e_2]\mathcal{R}')$$
$$\text{and } sizeof(e) < sizeof([e_1]\mathcal{R})$$

$$divide([e_1]\mathcal{R}, [e_2]\mathcal{R}') = split([e_1]\mathcal{R})$$

$$split([F]\mathcal{G}_1 : \ldots : \mathcal{G}_n : \epsilon) = ([x]\mathcal{G}_n : \epsilon, [([F]\mathcal{G}_1 : \ldots : \mathcal{G}_{n-1} : \epsilon)/x]) \qquad x \text{ fresh}$$

$$msg([e_1]\mathcal{R}_1, [e_2]\mathcal{R}_2) = ([e]\mathcal{R}, \theta\theta') \qquad\qquad\qquad\qquad \text{if } \mathcal{R}' = \epsilon$$
$$= ([x]\mathcal{R}', \theta\theta'[[e]\mathcal{R}/x]) \qquad\qquad \text{otherwise}$$
$$\text{if } (e, \theta) = msg(e_1, e_2) \text{ and } (\mathcal{R}, \mathcal{R}', \theta') = msg(\mathcal{R}_1, \mathcal{R}_2)$$

$$msg(\square\, e_1, \square\, e_2) = (\square\, e', \theta) \qquad\qquad\qquad\qquad\qquad \text{iff } (e', \theta) = msg(e_1, e_2)$$
$$msg(\square \oplus e_1, \square \oplus e_2) = (\square \oplus e', \theta) \qquad\qquad\qquad\quad \text{iff } (e', \theta) = msg(e_1, e_2)$$
$$msg(e_1 \oplus \square, e_2 \oplus \square) = (e' \oplus \square, \theta) \qquad\qquad\qquad\quad \text{iff } (e', \theta) = msg(e_1, e_2)$$
$$msg(\texttt{case } \square \texttt{ of } \{p_i \rightarrow e_{1i}\}, \texttt{case } \square \texttt{ of } \{p_i \rightarrow e_{2i}\}) = (\texttt{case } \square \texttt{ of } \{p_i \rightarrow e'_i\}, \theta_1 \ldots \theta_i) \quad \text{iff } (e'_i, \theta_i) = msg(e_{1i}, e_{2i})$$

$$msg(\lambda x.e_1, \lambda x.e_2) = (\lambda x.e', \theta) \qquad\qquad\qquad\qquad\quad \text{iff } (e', \theta) = msg(e_1, e_2)$$
$$msg(F, F) = (F, \emptyset)$$
$$msg(K, K) = (K, \emptyset)$$
$$msg(n, n) = (n, \emptyset)$$
$$msg(x, x) = (x, \emptyset)$$
$$msg(e_1, e_2) = (x, [e_1/x]) \qquad\qquad\qquad\qquad\qquad\qquad\quad x \text{ fresh}$$

$$msg(\mathcal{G}_1 : \mathcal{R}_1, \mathcal{G}_2 : \mathcal{R}_2) = (\mathcal{G} : \mathcal{R}, \mathcal{R}', \theta\theta'), \qquad \text{if } (\mathcal{G}, \theta) = msg(\mathcal{G}_1, \mathcal{G}_2) \text{ and } (\mathcal{R}, \mathcal{R}', \theta') = msg(\mathcal{R}_1, \mathcal{R}_2)$$

$$msg(\mathcal{R}_1, \mathcal{R}_2) = (\epsilon, \mathcal{R}_1, \emptyset) \qquad\qquad\qquad\qquad\qquad\qquad \text{(default)}$$

Figure 6. Divide, split and msg in zipper form

4.6 Discarding Expressions

The savings that are propagated by the supercompiler are used as input to a predicate *accept* which decides whether to discard or keep transformed functions. Our current implementation is remarkably simple and unconditionally discards expressions that are deemed too large, admit size reductions if they are accompanied with some reduction savings and require size increases to be accompanied by more reduction savings:

```
acceptable before after savings
 | sizeof(after) > 40 = False
 | sizeof(after) > 20 = savings − 100 <= 29
 | sizeof(before) >= sizeof(after) = s − 3 <= 0
 | otherwise = s − 50 <= 29
```

The reason this function works at all is that our algorithm will by construction emit a lot of small functions. If a function body contains many syntactic nodes it is a sign that optimization was not very successful. This typically happens in code that contains a lot of FFI-calls which are opaque to the supercompiler.

5. Measurements

The results supported by this section can be summarized with a single sentence: we can supercompile most programs from the spectral and imaginary part of nofib in less than two seconds of compile time while keeping the maximum binary size increase below 5%.

We start by establishing a base line against ghc -O2 on a small set of examples from the imaginary part of nofib as shown in Table 1. The table uses output from *nofib-analyse*, where each column indicates the difference between the baseline (O2 in this case) and the optimization. Absolute numbers without a sign, such as the runtime for rfib, means that the absolute numbers were too small to give reliable information. The size column is the size of the binary produced by ghc, which includes the runtime system and libraries. By turning off the features normalization of expressions,

boring contexts, discarding expressions and tainting expressions in our supercompiler we get to something that is close to an ordinary positive supercompiler [27]. The compilation times range from 2.5 seconds to almost 10 seconds.

We suffer from the same problem that Bolingbroke and Peyton Jones [3] report: the supercompiler can sometimes prevent GHC from applying other important optimizations such as unboxing of arithmetics. Since that is a slightly different algorithm it is not exactly the same programs that get staggering increases in memory allocations. A particular example is *tak*, which our supercompiler slightly optimizes, whereas their get an 18,000-fold increase in memory allocations.

The situation is the reverse with *rfib*, where we get +1599532.1% in allocations and their supercompiler does not change memory allocations at all. The optimized *rfib* program has a recursive function taking an unboxed double and returning an unboxed double, making the program run in constant memory and nearly constant allocations regardless of input size. The supercompiled program is not unboxed and the short recursive function that appears in the ordinary program is stretched out through numerous other function calls making the actual program flow very hard to follow for the human eye.

Looking closer at what happens inside the supercompiler when it produces this program is interesting: the whistle blows and the expression currently transformed is split into two parts. This split forces the inner part of the expression, which is the tight loop, to get a return type that is boxed. It is important to notice that even if our supercompiler used the standard versions of the homeomorphic embedding and the msg, the whistle and the split would produce the same results.

Integrate and *wheel-sieve2* are two other examples that shows the problem with our supercompiler preventing GHC from performing unboxing. The symptom is the same: a vast amount of increase in allocations for the supercompiled program.

There are two examples that show a rather hefty increase in binary size: *digits-of-e1* and *integrate*. One might argue that a 20%

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (s)
bernouilli	+8.0	-1.5	+0.0	+1.5	+0.0	3.52
digits-of-e1	+16.6	-20.3	-19.2	-16.0	+0.0	8.67
exp3_8	+6.0	-16.7	-17.6	-17.3	+0.0	2.58
integrate	+20.4	+65230.5	+104925.9	+93051.6	+30.4	9.02
primes	+6.0	+0.2	-13.9	-16.0	+0.0	2.46
rfib	+6.0	+1599532.1	0.18	0.18	+0.0	2.45
tak	+6.0	-12.5	-2.6	-2.8	+0.0	2.42
wheel-sieve2	+7.3	+307.0	0.54	+195.2	+252.1	3.15

Table 1. ghc -O2 compared with Supercompilation

reduction in runtime is worth a size increase of 16.6% for *digits-of-e1*. This is obviously not the case with *integrate*, a massive slowdown combined with more than 20% size increase of the resulting binary.

By turning on normalization of expressions (Table 2) there is a slight decrease of binary size, a small decrease of compilation time and no changes to the performance of the output programs. This is a small but worthwhile improvement over regular positive supercompilation.

Table 3 shows that by turning on boring contexts it is possible to further shrink the binary size and reduce compilation time. Two benchmarks are affected by this performance wise: *digits-of-e1* is optimized with another 20% reduction in memory allocations giving a slight performance improvement while *tak* goes back to the same amount of memory allocations as without supercompilation and a slight performance decrease.

By discarding expressions (Table 4) there is a further decrease of binary size and the performance loss *wheel-sieve2* suffered from plain supercompilation is somewhat mitigated. The compilation times are all shorter except for *digits-of-e1* which supercompiles several expressions and discards them only to find them in a slightly different context at a later time and repeating this work.

Tainting expressions (Table 5) gives an additional decrease of binary size and compilation with two benchmarks showing speedups: the performance of integrate is improved compared to ordinary optimization and *wheel-sieve2* is back to the same performance as without supercompilation. *Digits-of-e1* becomes slightly worse, but there is still a 16% reduction of memory compared to ordinary optimization and the compilation time is reduced to less than two seconds.

We finally give the full table with a greater selection of benchmarks from both the spectral and imaginary parts of nofib in Table 6. *Integrate* is the benchmark with the biggest size increase at 4.3%, but the supercompiler also manages to remove nearly one fifth of the total allocations.

Circsim is another interesting example, because it consists of about 400 lines of code and compiles in 1.12 seconds with the supercompiler turned on. The supercompiler manages to reduce the memory allocations slightly and give 0.6% binary size increase.

We saw that before adding the tainting of functions it was possible for the supercompiler to perform a lot of unnecessary work by speculatively supercompiling an expression, discard the result, supercompile the subexpressions, discard those results, and so on. One possible strategy to mitigate this further is to keep the supercompiled result around as a cache that successive supercompilation of subexpressions can fold against, thereby avoiding to repeat work.

Our experience so far is that the discarding of functions is not preventing optimizations, on the contrary: programs that were optimized by the supercompiler without discarding expressions are actually slightly faster when discarding the large useless specializations that ended up in the program previously.

6. Implementation Notes

This section contains an outline of the most obvious issues and areas for improvement that any programmer will come across when implementing a supercompiler.

6.1 Parallelism

The supercompiler is currently defined in a very sequential way: it is not possible to inspect the store unless the previous parts of the expression is supercompiled and the store contains all the new definitions. Beyond a lot of infrastructure hacking it appears that lifting this restriction while preserving code sharing is an interesting algorithmic problem that probably requires some research.

There are however some parts of the supercompiler as it is currently defined that are inherently parallel: the testing for folding and non-termination in $\mathcal{D}_{app}()$ can easily be done in parallel since the common case is that there are several possible candidates to test against. This gave a measurable speedup in an an earlier version of our prototype but had to be removed because other parts of GHC are not threadsafe.

6.2 Name Capture

Avoiding name capture can cause significant churn and make the supercompiler dreadfully slow. Bolingbroke and Peyton Jones [3] report that for a particular example 42% of their supercompilation time is spent on managing names and renaming.

Our current implementation uses eager substitutions with the scheme outlined by Peyton Jones and Marlow [19]. There are two obvious improvements to this: using a lazy substitution that the supercompiler applies as it traverses the expression would avoid some churn, and this could be further reduced by allowing some shadowing during supercompilation.

6.3 Representation Improvements for the Homeomorphic Embedding

Our implementation stores expressions annotated with their size in the memoization list. These annotations can be used for avoiding some testing altogether: an expression can only fold against an expression of equal size and an expression can never be homeomorphically embedded in a smaller expression. A similar representation was also independently suggested by Ilya Klyuchnikov in a personal conversation so this seems to be a well known optimization in the supercompilation community.

6.4 Splitting Typed Terms

The most obvious difference between our implementation and the algorithm described in this paper is that the intermediate language of GHC is a typed language [28]. This brings additional complexity when generalizing similar terms of different type against eachother.

Pfenning [20] has already identified our problems: the presence of binders in expressions and the presence of types in binders. There are two advantages for our particular setting: 1) type vari-

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-1.6	+0.0	+0.0	-2.2	+0.0	-27.8
digits-of-e1	-1.6	+0.0	-0.5	+0.5	+0.0	-10.5
exp3_8	-1.6	+0.0	-0.9	+1.6	+0.0	-33.7
integrate	-1.5	+0.1	-0.3	-0.4	-3.0	-6.8
primes	-1.8	+0.0	+0.0	+0.0	+0.0	-32.9
rfib	-1.6	+0.0	0.18	0.18	+0.0	-37.5
tak	-1.6	+0.0	+0.0	+0.5	+0.0	-38.8
wheel-sieve2	-1.7	+0.0	-0.4	-1.3	+0.0	-29.5

Table 2. Supercompilation compared with Supercompilation+normalization

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-2.1	+0.0	+0.8	-1.5	+0.0	-30.3
digits-of-e1	-1.5	-22.6	-4.9	-7.4	+0.0	-10.7
exp3_8	-1.6	+0.0	+0.9	-3.2	+0.0	-40.3
integrate	-3.9	+0.3	+0.9	+1.0	+2.3	-21.2
primes	-1.5	+0.0	+0.0	+5.1	+0.0	-40.6
rfib	-1.6	+0.0	0.16	0.17	+0.0	-38.6
tak	-1.6	+13.9	+16.0	+15.6	+0.0	-37.8
wheel-sieve2	-1.6	+0.0	+0.0	+0.0	+0.0	-33.4

Table 3. Supercompilation+normalization compared with Supercompilation+normalization+boring expressions

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-1.7	+0.0	-0.8	-0.8	+0.0	-35.0
digits-of-e1	-1.0	+0.0	-2.6	+0.5	+0.0	+78.6
exp3_8	-2.1	+0.0	+0.0	+0.0	+0.0	-43.1
integrate	-4.0	-0.1	-0.2	-0.2	+0.8	-9.5
primes	-1.9	+0.0	+0.0	-4.8	+0.0	-42.9
rfib	-2.0	+0.0	0.18	0.18	+0.0	-46.8
tak	-2.0	+0.0	-21.8	-21.8	+0.0	-45.7
wheel-sieve2	-1.9	-73.6	-64.8	-65.0	-72.2	-33.1

Table 4. Supercompilation+normalization+boring expressions compared with Supercompilation+normalization+boring expressions+discarding expressions

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-1.2	-0.8	+0.0	+3.1	+0.0	-44.3
digits-of-e1	-8.9	+8.7	+2.6	-0.5	+0.0	-89.6
exp3_8	-0.3	+0.0	+0.0	+0.0	+0.0	-32.8
integrate	-4.7	-99.9	-99.9	-99.9	-23.3	-64.8
primes	-0.4	+0.0	-3.2	+0.0	+0.0	-35.7
rfib	-0.4	+0.0	0.18	0.19	+0.0	-36.0
tak	-0.4	+0.0	+5.9	+5.9	+0.0	-36.0
wheel-sieve2	-1.5	-6.9	0.19	-1.9	+2.1	-52.5

Table 5. Supercompilation+normalization+boring expressions+discarding expressions compared with Supercompilation+normalization+boring expressions+discarding expressions+tainting functions

ables are the top element for types; and 2) it is safe to fail during generalization and call split on the term instead.

After having implemented the type generalization we noticed that it was not performed very often in practice on our examples from nofib. The main cost related to this feature appears to be the implementation cost.

7. Related Work

Turchin [29] invented supercompilation in the context of the functional language Refal. The supercompiler Scp4 [17] is implemented in Refal and is the most well-known implementation from this line of work. Sørensen et al. [27] and Secher [23], Secher and Sørensen [24] presented supercompilers formulated for pure first order functional languages but never reported anything about the performance of the algorithms or the resulting programs.

More recently there has been a spur of activity in actual implementations of supercompilers:

Supero Supero [14, 16] could optimize a text book example of word counting written in Haskell to the same performance as a C program performing the same task. The latest version has changed the design significantly in order to improve performance of the supercompiler and Mitchell [15] report compilation times below 4 seconds for all examples.

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (s)
bernouilli	+1.0	-2.3	+0.0	-2.9	+0.0	0.64
digits-of-e1	+1.9	-32.9	-23.5	-21.9	+0.0	1.29
exp3_8	+0.1	-16.7	-17.6	-19.8	+0.0	0.39
integrate	+4.3	-18.7	-0.6	-2.6	+0.0	2.11
primes	+0.1	+0.2	-17.1	-17.4	+0.0	0.36
rfib	+0.1	+1599533.0	0.18	0.19	+0.0	0.32
tak	+0.1	-0.3	-6.5	-6.5	+0.0	0.32
wheel-sieve2	+0.1	+0.0	0.19	+0.0	+0.0	0.47
atom	+3.0	-32.3	-49.8	-49.4	-33.3	1.50
circsim	+0.6	-1.5	-2.3	-2.6	-4.8	1.12
clausify	+0.6	-1.9	-0.7	-0.7	+0.0	0.71
constraints	+0.7	+14.3	-33.7	-33.9	-96.9	0.80
cryptarithm1	+0.3	-7.5	-5.6	-3.4	+0.0	0.38
eliza	+1.2	-40.8	0.00	0.00	+0.0	1.09
fish	+2.1	-4.9	0.01	0.01	+0.0	0.92
gcd	+0.6	-5.8	0.09	0.10	+0.0	0.51
lcss	+1.4	+10.3	+8.8	+10.2	-11.1	0.82
life	+0.4	+1.2	0.13	0.14	+0.0	0.52
mandel2	+1.3	+4.5	0.00	0.00	+0.0	0.99
minimax	+0.1	+0.0	0.00	0.00	+0.0	0.30
pretty	+0.6	-3.3	0.00	0.00	+0.0	0.54
primetest	+3.6	-0.2	+0.0	+0.0	+0.0	1.58

Table 6. ghc -O2 compared with Supercompilation with all features enabled

CHSC Bolingbroke and Peyton Jones [3] presented a supercompiler with call-by-need semantics that did not require its input to be lambda lifted.

HOSC Klyuchnikov and Romanenko [9, 10] have implemented HOSC in Scala and used it for proving equivalence of terms.

Others Riech et al. [21] use supercompilation to optimize programs intended to run on the Reduceron, an FPGA-based soft processor for executing lazy functional programs. They report average performance boosts on a set of programs of 40%. Supercompilation has also been investigated in a call-by-value setting and can give good speedups there too [6, 8].

Narrowing-driven partial evaluation [1, 2] is the functional logic programming equivalent of positive supercompilation but formulated as a term rewriting system. Partial evaluators such as Ecce and Logen [13] seem to be more popular approaches in the logic programming community and more recent work has focused on scaling partial evaluation so that it can be applied to large programs [12, 30].

8. Future Work

Supercompilation is still a rather expensive optimization, but it can undoubtedly produce great results on high level Haskell programs. The major obstacle we see at the moment is that supercompilation might prevent the compiler from applying other optimizations, which in turn can increase the runtime of the final executable by several orders of magnitude. There are several possible ways to tackle this problem:

- Integrate other optimizations into the supercompiler

- Make the supercompiler output programs that are friendly to other optimizations

- Strengthen other optimizations so that they can optimize the output from the supercompiler

The approximation of boring expressions could be extended in several ways to avoid transforming expressions that will give bad results. One observation is that the first call to reverse with an accumulating parameter ($reverse\ xs\ []$) is not boring since it has a known constructor in the parameter, but by inspecting the body of $reverse$ it is obvious that this parameter is never used. One idea is to extend boring expressions to utilize strictness information so that it would discard this example because the static information present in the context is never used. This idea is related to the concept of "deterministic unfolding" in logic programming.

There is certainly room to improve the decision whether to discard supercompiled expressions. The constants currently used could be tuned further, and there is also the possibility to use other information that is present in the program.

A route that we have not explored is to use different acceptable-predicates for different parts of the supercompiler. Rule R19 and R20 could be extended with a similar test which could allow that only some parts of $\mathcal{R}$ was pushed down.

We observed that there are sometimes several candidates in the memoization list that the supercompiler could generalize the current expression against. Our current implementation selects the first one which is not necessarily the best candidate. A ranking function between different candidates would be useful.

Bolingbroke and Peyton Jones [3] have presented a call-by-need algorithm for supercompilation. We believe that the algorithm presented in this paper could be adjusted to call-by-need semantics as well.

Leuschel [11] has investigated interesting variations of the homeomorphic embedding and we are currently investigating whether similar proof techniques can be used for our whistle.

The focus in this paper has been on a practical algorithm that can supercompile programs without risking code explosion. We believe that our previous proofs of correctness of a closely related algorithm [7] should go through without much adjustment for the algorithm presented.

9. Conclusions

We have presented a supercompilation algorithm and shown that it can give significant speedups on programs from the nofib suite even under a tight code budget constraint. We have also shown that larger programs are supercompiled in reasonable time and also

benefits from the aggressive inlining and specialization that our supercompiler performs.

Acknowledgments

References

[1] E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Comput*, 20(1):3–26, 2001.

[2] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[3] M. Bolingbroke and S. L. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 135–146. ACM, 2010.

[4] G. Huet. The zipper. *Journal of Functional Programming*, 7(05): 549–554, 1997.

[5] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[6] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language. In *POPL*, pages 277–288, 2009.

[7] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language. *Logical Methods in Computer Science*, 6(3), 2010.

[8] P. A. Jonsson and J Nordlander. Strengthening Supercompilation for Call-By-Value Languages. In *Second International Workshop on Metacomputation in Russia*, pages 64–81, 2010.

[9] I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *PSI '09: Proceedings of the Seventh International Andrei Ershov Memorial Conference*, 2009.

[10] I. Klyuchnikov and S. Romanenko. Towards Higher-Level Supercompilation. In *Second International Workshop on Metacomputation in Russia*, pages 82–101, 2010.

[11] M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In T. Æ. Mogensen, D. A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 379–403. Springer, 2002. ISBN 3-540-00326-6.

[12] M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'08)*, pages 119–134. Springer LNCS 5438, 2009.

[13] M. Leuschel, D. Elphick, M. Varea, S-J Craig, and M. Fontaine. The ecce and logen partial evaluators and their web interfaces. In *PEPM*, pages 88–94, New York, NY, USA, 2006. ACM. ISBN 1-59593-196-1.

[14] N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, June 2008.

[15] N. Mitchell. Rethinking supercompilation. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 309–320. ACM, September 2010. ISBN 978-1-60558-794-3.

[16] N. Mitchell and C. Runciman. A supercompiler for core haskell. In O. Chitil et al., editor, *Selected Papers from the Proceedings of IFL 2007*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer-Verlag, 2008.

[17] A. P. Nemytykh. The supercompiler SCP4: General structure. In M. Broy and A. V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Revised Papers*, volume 2890 of *LNCS*, pages 162–170. Springer, 2003. ISBN 3-540-20813-5.

[18] W. Partain. The nofib benchmark suite of haskell programs. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992. ISBN 3-540-19820-2.

[19] S. L. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program*, 12(4&5):393–433, 2002.

[20] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

[21] J. S Riech, M. Naylor, and C. Runciman. Supercompilation and the Reduceron. In *Second International Workshop on Metacomputation in Russia*, pages 159–172, 2010.

[22] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1–2):193–233, 30 October 1996.

[23] J. P. Secher. Perfect supercompilation. Technical Report DIKU-TR-99/1, Department of Computer Science (DIKU), University of Copenhagen, February 1999.

[24] J.P. Secher and M.H. Sørensen. On perfect supercompilation. In D. Bjørner, M. Broy, and A. Zamulin, editors, *Proceedings of Perspectives of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, 2000.

[25] M.H. Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program*, 37(1-3):163–205, 2000.

[26] M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *International Logic Programming Symposium*, pages 465–479. Cambridge, MA: MIT Press, 1995.

[27] M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[28] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international worksh op on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X.

[29] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[30] G. Vidal. A Hybrid Approach to Conjunctive Partial Deduction. In *Proc. of the 21th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'10)*, 2010. Available from http://users.dsic.upv.es/~gvidal/german/papers.html.

[31] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. ISSN 0304-3975.