Positive Supercompilation for a Higher Order Call-By-Value Language

Peter A. Jonsson Johan Nordlander
Luleå University of Technology
{pj, nordland}@csee.ltu.se

Abstract

Previous deforestation and supercompilation algorithms may introduce accidental termination when applied to call-by-value programs. This hides looping bugs from the programmer, and changes the behavior of a program depending on whether it is optimized or not. We present a supercompilation algorithm for a higher-order call-by-value language and we prove that the algorithm both terminates and preserves termination properties. This algorithm utilizes strictness information for deciding whether to substitute or not and compares favorably with previous call-by-name transformations.

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

General Terms Languages, Theory

Keywords supercompilation, deforestation, call-by-value

1. Introduction

Intermediate lists in functional programs allow the programmer to write clear and concise programs, but carry a cost at run time since list cells need to be both allocated and garbage collected. Both deforestation (Wadler 1990) and supercompilation (Sørensen et al. 1996) are automatic program transformations which remove many of these intermediate structures. In a call-by-value context these transformations are unsound, and might hide looping bugs from the programmer. Consider the program

$$(\lambda x.y)\,(3/z).$$

This program could contain a division by zero, if the value of z is zero. Applying Wadler's deforestation algorithm to the program will result in y, which is sound under call-by-name or call-by-need. Under call-by-value the division by zero in the original program has been removed, and hence the meaning of the program has been altered by the transformation.

Removing intermediate structures in a call-by-value language is perhaps even more important than in a lazy language since the entire intermediate structure has to stay alive during the computation.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

Ohori and Sasano (2007) saw this need and presented a very elegant algorithm, for call-by-value languages, that removes intermediate structures. Their algorithm sacrifices some transformational power for algorithmic simplicity. We explore a different part of the design space: a more powerful transformation at the cost of some algorithmic complexity. We show how to construct a meaning-preserving supercompiler for pure call-by-value languages in general and implement it in a compiler for a pure call-by-value language (Nordlander et al. 2008).

This is a necessary first step towards supercompiling impure call-by-value languages, of which there are many readily available today. Well known examples are OCaml (Leroy 2008), Standard ML (Milner et al. 1997) and F# (Syme 2008). Considering that F# is currently being turned into a product it is quite likely that strict functional languages will be even more popular in the future.

One might think that our result should be easily obtainable by modifying a call-by-name algorithm to simply delay beta-reduction until every function argument has been specialized to a value. However, it turns out that this strategy misses even simple opportunities to remove intermediate structures. The explanation is that eager specialization of function arguments risks destroying *fold* opportunities that might otherwise appear, something which may even prohibit complexity improvements to the resulting program.

The novelty of our supercompilation algorithm is that it concentrates all call-by-value dependencies to a single rule that relies on the result from a separate strictness analysis for correct behavior. In effect, our algorithm delays transformation of function arguments past inlining, much like a call-by-name scheme does, although only as far as allowed by call-by-value semantics. The result is an algorithm that is able to improve a wide range of illustrative examples like the existing algorithms do, but without the risk of introducing artificial termination.

The specific contributions of our work are:

- We provide an algorithm for positive supercompilation including folding, for a strict and pure higher-order functional language (Section 4).
- We prove that the algorithm terminates and preserves the semantics of the program (Section 5).
- We show preliminary benchmarks from an implementation in the Timber compiler (Section 6).
- We outline variations of the algorithm that makes it perform better for certain programs (Section 7).

We start out with some examples in Section 2 to give the reader an intuitive feel of how the algorithm behaves. Our language of study is defined in Section 3, right before the technical contributions are presented.

2. Examples

Wadler (1990) uses the example $append\ (append\ xs\ ys)\ zs$ and shows that his deforestation algorithm transforms the program so that it saves one traversal of the first list, thereby reducing the complexity from $2|xs| + |ys|$ to $|xs| + |ys|$.

$$\begin{aligned} append\ xs\ ys = {} & \mathbf{case}\ xs\ \mathbf{of} \\ & [] \ \rightarrow\ ys \\ & (x : xs) \ \rightarrow\ x : append\ xs\ ys \end{aligned}$$

If we naïvely change Wadler's algorithm to call-by-value semantics by eagerly attempting to transform arguments before attacking the body, we do not achieve this improvement in complexity. An example from a hypothetical deforestation algorithm that attacks arguments first is:

$$f\ =\ append\ (append\ xs'\ ys')\ zs'$$

(Inline the body of $append\ xs'\ ys'$ in the context $append\ []\ zs'$ and push down the context into each branch of the case)

$$\begin{aligned} f\ = {} & \mathbf{case}\ xs'\ \mathbf{of} \\ & [] \ \rightarrow\ append\ ys'\ zs' \\ & (x : xs) \ \rightarrow\ append\ (x : append\ xs\ ys')\ zs' \end{aligned}$$

(Transform each branch but focus on the (*x:xs*) case)

$$append\ (x : append\ xs\ ys')\ zs'$$

(The expression contains an expression that has already been seen. The subexpression $x : append\ xs\ ys'$ is extracted for separate transformation)

$$\begin{aligned} & x : append\ xs\ ys' \\ & x :\ \mathbf{case}\ xs\ \mathbf{of} \\ & \quad [] \ \rightarrow\ ys' \\ & \quad (x' : xs') \ \rightarrow\ x' : append\ xs'\ ys' \end{aligned}$$

(A renaming of a previous expression in the second branch)

The end result from this transformation is:

$$\begin{aligned} f\ = {} & \mathbf{case}\ xs'\ \mathbf{of} \\ & [] \ \rightarrow\ h_1\ ys'\ zs' \\ & (x : xs) \ \rightarrow\ h_1\ (h_2\ x\ xs\ ys')\ zs' \\ h_1\ xs\ ys\ = {} & \mathbf{case}\ xs\ \mathbf{of} \\ & [] \ \rightarrow\ ys \\ & (x' : xs') \ \rightarrow\ x' : h_1\ xs'\ ys \\ h_2\ x\ xs\ ys = {} & x :\ \mathbf{case}\ xs\ \mathbf{of} \\ & [] \ \rightarrow\ ys \\ & (x' : xs') \ \rightarrow\ h_2\ x'\ xs'\ ys \end{aligned}$$

The intermediate structure from the input program is still there after the transformation, and the complexity remains at $2|xs| + |ys|$!

However, doing the exact opposite — that is, carefully delaying transformation of arguments to a function past inlining of its body — actually leads to the same result as Wadler obtains after transforming $append\ (append\ xs\ ys)\ zs$. This is a key observation for obtaining deforestation under call-by-value without altering semantics, and our algorithm exploits it.

Except for this key difference, which is necessary to preserve semantics, our algorithm shares many of its rules with Wadler's algorithm. The transformation that is commonly referred to as case-of-case is crucial for our algorithm, just like it is for a call-by-name algorithm. The case-of-case transformation is useful when a case statement appears in the head of another case statement, in which case the outer case statement is duplicated and pushed into all branches of the inner case statement. Our algorithm also contains rules that correspond to ordinary evaluation which eliminate case statements that have a known constructor in their head or to add two primitive numbers. The mechanism that ensures termination basically looks for "similar" terms to ones that have already been transformed, and if that happens will stop the transformation by splitting the term into smaller terms that are transformed separately. The remaining rules of our algorithm shifts the focus to the proper subexpression of expressions and ensures the algorithm does not get stuck.

We claim that our algorithm compares favorably with previous call-by-name transformations, and proceed with demonstrating the transformation of common examples. The results are equal to those of Wadler (1990). Our first example is transformation of $sum\ (map\ square\ ys)$. The functions used in the examples are defined as:

$$\begin{aligned} square\ x\ = {} & x * x \\ map\ f\ xs = {} & \mathbf{case}\ xs\ \mathbf{of} \\ & [] \ \rightarrow\ ys \\ & (x : xs) \ \rightarrow\ f\ x : map\ f\ xs \\ sum\ xs\ \ = {} & \mathbf{case}\ xs\ \mathbf{of} \\ & [] \ \rightarrow\ 0 \\ & (x : xs) \ \rightarrow\ x\ +\ sum\ xs \end{aligned}$$

We start our transformation by allocating a new fresh function name (h_0) to this expression, inlining the body of sum and substituting $map\ square\ ys$ into the body of sum:

$$\begin{aligned} & \mathbf{case}\ map\ square\ ys\ \mathbf{of} \\ & \quad [] \ \rightarrow\ 0 \\ & \quad (x' : xs') \ \rightarrow\ x'\ +\ sum\ xs' \end{aligned}$$

After inlining map and substituting the arguments into the body the result becomes:

$$\begin{aligned} & \mathbf{case}\ (\ \mathbf{case}\ ys\ \mathbf{of} \\ & \quad [] \ \rightarrow\ [] \\ & \quad (x' : xs') \ \rightarrow\ (square\ x') : map\ square\ xs')\ \mathbf{of} \\ & \quad [] \ \rightarrow\ 0 \\ & \quad (x' : xs') \ \rightarrow\ x'\ +\ sum\ xs' \end{aligned}$$

We duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch with ordinary reduction steps yields:

$$\begin{aligned} & \mathbf{case}\ ys\ \mathbf{of} \\ & \quad [] \ \rightarrow\ 0 \\ & \quad (x' : xs') \ \rightarrow\ square\ x'\ +\ sum\ (map\ square\ xs') \end{aligned}$$

Now inline the body of the first square and observe that the second argument to $(+)$ is similar to the expression we started with. We replace the second parameter to $(+)$ with $h_0\ xs'$. The result of our transformation is $h_0\ ys$, with h_0 defined as:

$$\begin{aligned} h_0\ ys = {} & \mathbf{case}\ ys\ \mathbf{of} \\ & [] \ \rightarrow\ 0 \\ & (x' : xs') \ \rightarrow\ x' * x'\ +\ h_0\ xs' \end{aligned}$$

This new function only traverses its input once, and no intermediate structures are created. If the expression *sum (map square xs)* or a renaming thereof is detected elsewhere in the input, a call to h_0 will be inserted there instead.

The work by Ohori and Sasano (2007) can not fuse two successive applications of the same function, nor mutually recursive functions. We show that our algorithm can handle these two cases. We need the following new function definitions:

$$
\begin{array}{lll}
mapsq\ xs = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & [] & \to\ [] \\
 & (x' : xs') & \to\ (x' * x') : mapsq\ xs' \\
f\ xs = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & [] & \to\ [] \\
 & (x' : xs') & \to\ (2 * x') : g\ xs' \\
g\ xs = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & [] & \to\ [] \\
 & (x' : xs') & \to\ (3 * x') : f\ xs'
\end{array}
$$

Transforming $mapsq\ (mapsq\ xs)$ will inline the outer $mapsq$, substitute the argument in the function body and inline the inner call to $mapsq$:

$$
\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of} \\
\qquad [] \to [] \\
\qquad (x' : xs') \to (x' * x') : mapsq\ xs')\ \textbf{of} \\
[] \to [] \\
(x' : xs') \to (x' * x') : mapsq\ xs'
\end{array}
$$

As previously, we duplicate the outer case in each of the inner case's branches, using the expression in the branches as head of that case-statement. Continuing the transformation on each branch by ordinary reduction steps yields:

$$
\begin{array}{l}
\textbf{case}\ xs\ \textbf{of} \\
[] \to [] \\
(x' : xs') \to (x' * x' * x' * x') : mapsq\ (mapsq\ xs')
\end{array}
$$

This will encounter a similar expression to what we started with, and create a new function h_1. The final result of our transformation is $h_1\ xs$, with the new residual function h_1 that only traverses its input once defined as:

$$
\begin{array}{lll}
h_1\ xs = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & [] & \to\ [] \\
 & (x' : xs') & \to\ (x' * x' * x' * x') : h_1\ xs'
\end{array}
$$

For an example of transforming mutually recursive functions, consider the transformation of $sum\ (f\ xs)$. Inlining the body of sum, substituting its arguments in the function body and inlining the body of f yields:

$$
\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of} \\
\qquad [] \to [] \\
\qquad (x' : xs') \to (2 * x') : g\ xs')\ \textbf{of} \\
[] \to 0 \\
(x' : xs') \to x' + sum\ xs'
\end{array}
$$

We now move down the outer case into each branch, and perform reductions until we end up with:

$$
\textbf{case}\ xs\ \textbf{of}\ \{[] \to 0;\ (x' : xs') \to (2 * x') + sum\ (g\ xs')\ \}
$$

We notice that unlike in previous examples, $sum\ (g\ xs')$ is not similar to what we started transforming. For space reasons, we focus on the transformation of the rightmost expression in the last branch, $sum\ (g\ xs')$, while keeping the functions already seen in mind. We inline the body of sum, perform the substitution of its arguments and inline the body of g:

$$
\begin{array}{l}
\textbf{case}\ (\ \textbf{case}\ xs'\ \textbf{of} \\
\qquad [] \to [] \\
\qquad (x'' : xs'') \to (3 * x'') : f\ xs'')\ \textbf{of} \\
[] \to 0 \\
(x' : xs') \to x' + sum\ xs'
\end{array}
$$

We now move down the outer case into each branch, and perform reductions:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
[] \to 0 \\
(x'' : xs'') \to (3 * x'') + sum\ (f\ xs'')
\end{array}
$$

Expressions

$$
\begin{array}{lll}
e,\ f & ::= & n \mid x \mid g \mid f\,\overline{e} \mid \lambda\overline{x}.e \mid k\,\overline{e} \mid e_1 \oplus e_2 \\
 & \mid & \textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\} \mid \textbf{let}\ x = f\ \textbf{in}\ e \\
 & \mid & \textbf{letrec}\ g = v\ \textbf{in}\ e \\
p & ::= & n \mid k\,\overline{x}
\end{array}
$$

Values

$$
v \quad ::= \quad n \mid \lambda\overline{x}.e \mid k\,\overline{v}
$$

Figure 1. The language

We notice a familiar expression in $sum\ (f\ xs'')$, and fold when reaching it. Adding it all together gives a new function h_2:

$$
\begin{array}{lll}
h_2\ xs = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & [] & \to\ 0 \\
 & (x' : xs') & \to\ (2 * x') + \textbf{case}\ xs'\ \textbf{of} \\
 & & \qquad [] \to 0 \\
 & & \qquad (x'' : xs'') \to \\
 & & \qquad\quad (3 * x'') + h_2\ xs''
\end{array}
$$

Kort (1996) studied a ray-tracer written in Haskell, and identified a critical function in the innermost loop of a matrix multiplication, called $vecDot$:

$$
vecDot\ xs\ ys = sum\ (zipWith\ (*)\ xs\ ys)
$$

This is simplified by our positive supercompiler to:

$$
\begin{array}{lll}
vecDot\ xs\ ys = & h_1\ xs\ ys & \\
h_1\ xs\ ys = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & (x' : xs') & \to\ \textbf{case}\ ys\ \textbf{of} \\
 & & \qquad (y' : ys') \to \\
 & & \qquad\quad x' * y' + h_1\ xs'\ ys' \\
 & & \qquad \_ \to 0 \\
 & \_ & \to\ 0
\end{array}
$$

The intermediate list between *sum* and *zipWith* is transformed away, and the complexity is reduced from $2|xs|+|ys|$ to $|xs|+|ys|$ (since this is matrix multiplication $|xs| = |ys|$).

3. Language

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

We let constructor symbols be denoted by *k*. Let g range over a set $\mathcal{G}$ of global definitions whose right-hand sides are all values.

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 $\oplus$.

We abbreviate a list of expressions $e_1 \ldots e_n$ as $\overline{e}$, and a list of variables $x_1 \ldots x_n$ as $\overline{x}$. All functions have a specific arity and all applications must be saturated; hence $\lambda x.map\ (\lambda y.y\ +\ 1)\ x$ is legal whereas $map\ (\lambda y.y\ +\ 1)$ is not.

We denote the free variables of an expression *e* by $fv(e)$, as defined in Figure 2. Along the same lines we denote the function names in an expression *e* as $fn(e)$, defined in Figure 3.

A program is an expression with no free variables and all function names defined in $\mathcal{G}$. The intended operational semantics is given in Figure 4, where $[\overline{e}/\overline{x}]e'$ is the capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e'.

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

$$
\begin{array}{lcl}
\mathit{fv}(x) & = & \{x\} \\
\mathit{fv}(n) & = & \emptyset \\
\mathit{fv}(g) & = & \emptyset \\
\mathit{fv}(k\,\overline{e}) & = & \mathit{fv}(\overline{e}) \\
\mathit{fv}(\lambda\overline{x}.e) & = & \mathit{fv}(e)\backslash\{\overline{x}\} \\
\mathit{fv}(f\,\overline{e}) & = & \mathit{fv}(f) \cup \mathit{fv}(\overline{e}) \\
\mathit{fv}(\textbf{let}\ x = e\ \textbf{in}\ f) & = & \mathit{fv}(e) \cup (\mathit{fv}(f)\backslash\{x\}) \\
\mathit{fv}(\textbf{letrec}\ g = v\ \textbf{in}\ f) & = & \mathit{fv}(v) \cup \mathit{fv}(f) \\
\mathit{fv}(\textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\}) & = & \mathit{fv}(e) \cup (\bigcup(\mathit{fv}(e_i)\backslash\mathit{fv}(p_i)) \\
\mathit{fv}(e_1 \oplus e_2) & = & \mathit{fv}(e_1) \cup \mathit{fv}(e_2)
\end{array}
$$

Figure 2. Free variables of an expression

$$
\begin{array}{lcl}
\mathit{fn}(x) & = & \emptyset \\
\mathit{fn}(n) & = & \emptyset \\
\mathit{fn}(g) & = & \{g\} \\
\mathit{fn}(k\,\overline{e}) & = & \mathit{fn}(\overline{e}) \\
\mathit{fn}(\lambda\overline{x}.e) & = & \mathit{fn}(e) \\
\mathit{fn}(f\,\overline{e}) & = & \mathit{fn}(f) \cup \mathit{fn}(\overline{e}) \\
\mathit{fn}(\textbf{let}\ x = e\ \textbf{in}\ f) & = & \mathit{fn}(e) \cup \mathit{fn}(f) \\
\mathit{fn}(\textbf{letrec}\ g = v\ \textbf{in}\ f) & = & (\mathit{fn}(v) \cup \mathit{fn}(f))\backslash\{g\} \\
\mathit{fn}(\textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\}) & = & \mathit{fn}(e) \cup (\bigcup(\mathit{fn}(e_i)) \\
\mathit{fn}(e_1 \oplus e_2) & = & \mathit{fn}(e_1) \cup \mathit{fn}(e_2)
\end{array}
$$

Figure 3. Function names of an expression

Reduction contexts

$$
\begin{array}{lcl}
\mathcal{E} & ::= & [\,] \mid \mathcal{E}\,\overline{e} \mid (\lambda\overline{x}.e)\,\overline{\mathcal{E}} \mid k\,\overline{\mathcal{E}} \mid \mathcal{E} \oplus e \mid n \oplus \mathcal{E} \\
& \mid & \textbf{case}\ \mathcal{E}\ \textbf{of}\ \{p_i \to e_i\} \mid \textbf{let}\ x = \mathcal{E}\ \textbf{in}\ e
\end{array}
$$

Evaluation relation

$$
\begin{array}{lcll}
\mathcal{E}\langle g\rangle & \mapsto & \mathcal{E}\langle v\rangle & \text{(Global)} \\
 & & \text{if } (g = v) \in \mathcal{G} & \\
\mathcal{E}\langle(\lambda\overline{x}.e)\,\overline{v}\rangle & \mapsto & \mathcal{E}\langle[\overline{v}/\overline{x}]e\rangle & \text{(App)} \\
\mathcal{E}\langle\textbf{let}\ x = v\ \textbf{in}\ e\rangle & \mapsto & \mathcal{E}\langle[v/x]e\rangle & \text{(Let)} \\
\mathcal{E}\langle\textbf{case}\ k\,\overline{v}\ \textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}\rangle & \mapsto & \mathcal{E}\langle[\overline{v}/\overline{x}_j]e_j\rangle & \text{(KCase)} \\
 & & \text{if } k = k_j & \\
\mathcal{E}\langle\textbf{case}\ n\ \textbf{of}\ \{n_i \to e_i\}\rangle & \mapsto & \mathcal{E}\langle e_j\rangle & \text{(NCase)} \\
 & & \text{if } n = n_j & \\
\mathcal{E}\langle n_1 \oplus n_2\rangle & \mapsto & \mathcal{E}\langle n\rangle & \text{(Arith)} \\
 & & \text{if } n = n_1 + n_2 &
\end{array}
$$

Figure 4. Reduction semantics

denotes a list of terms with just a single hole, evaluated from left to right.

If a variable appears no more than once in a term, that term is said to be *linear* with respect to that variable. Like Wadler (1990), we extend the definition slightly for linear **case** terms: no variable may appear in both the selector and a branch, although a variable may appear in more than one branch. The definition of *append* is linear, although ys appears in both branches.

We encode letrec as an application containing fix, where fix is defined as

$$fix = \lambda f.f\,(\lambda n.fix\,f\,n)$$

Definition 3.1. *Letrec is defined as:*

$$\textbf{letrec}\ h = \lambda\overline{x}.e\ \textbf{in}\ e' \stackrel{def}{=} (\lambda h.e')\,(\lambda y.fix\,(\lambda h.\lambda\overline{x}.e)\,y)$$

$$
\begin{array}{lcl}
\mathit{strict}(x) & = & \{x\} \\
\mathit{strict}(n) & = & \emptyset \\
\mathit{strict}(g) & = & \emptyset \\
\mathit{strict}(k\,\overline{e}) & = & \mathit{strict}(\overline{e}) \\
\mathit{strict}(\lambda\overline{x}.e) & = & \emptyset \\
\mathit{strict}(f\,\overline{e}) & = & \mathit{strict}(f) \cup \mathit{strict}(\overline{e}) \\
\mathit{strict}(\textbf{let}\ x = e\ \textbf{in}\ f) & = & \mathit{strict}(e) \cup (\mathit{strict}(f)\backslash\{x\}) \\
\mathit{strict}(\textbf{letrec}\ g = v\ \textbf{in}\ f) & = & \mathit{strict}(f) \\
\mathit{strict}(\textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\}) & = & \mathit{strict}(e) \cup (\bigcap(\mathit{strict}(e_i)\backslash\mathit{fv}(p_i)) \\
\mathit{strict}(e_1 \oplus e_2) & = & \mathit{strict}(e_1) \cup \mathit{strict}(e_2)
\end{array}
$$

Figure 6. The strict variables of an expression

By defining letrec as syntactic sugar for other primitives we introduce an implicit requirement that the right hand side of letrec statements must not contain any free variables except h. This is not a limitation since functions that contain free variables can be lambda lifted (Johnsson 1985) to the top level.

4. Higher Order Positive Supercompilation

It is time to make the intuition developed in Section 2 more formal. Our supercompiler is defined as a set of rewrite rules that pattern-match on expressions. This algorithm is called the *driving* algorithm, and is defined in Figure 5. Two additional parameters appear as subscripts to the rewrite rules: a memoization list ρ and a driving context $\mathcal{R}$. The memoization list holds information about expressions already traversed and is explained more in detail in Section 4.1. The driving context $\mathcal{R}$ is smaller than $\mathcal{E}$, and is defined as follows:

$$\mathcal{R} ::= [\,] \mid \mathcal{R}\,\overline{e} \mid \textbf{case}\,\mathcal{R}\ \textbf{of}\ \{p_i \to e_i\} \mid \mathcal{R} \oplus e \mid e \oplus \mathcal{R}$$

Interestingly this definition coincides with the evaluation contexts for a call-by-name language. The reason our algorithm still preserves a call-by-value semantics is that beta-reduction (rule R10) results in a let-binding, whose further specialization in rule R14 depends on whether the body expression f is strict in the bound variable x or not.

In principle, an expression e is strict with regards to a variable x if it eventually evaluates x; in other words, if $e \mapsto \ldots \mapsto \mathcal{E}\langle x\rangle$. Such information is not computable in general, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 6, where the strict variables of an expression e are defined as all free variables of e except those that only appear under a lambda or not inside all branches of a case.

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R11 and R20 are the default fall-back cases which extend the given driving context $\mathcal{R}$ and zoom in on the next expression to be driven. The program is turned "inside-out" by moving the surrounding context $\mathcal{R}$ into all branches of the case-statement through rules R16 and R19. Rule R14 has a similar mechanism for let-statements. Notice how the context is moved out of the recursive call in rule R5, whereas rule R7 recursively applies the driving algorithm to the full new term $\mathcal{R}\langle n\rangle$, forcing a re-traversal of the new term in hope of further reductions. Meta-variable a in rule R8 and rule R19 stands for an "annoying" expression; i.e., an expression that would be further reducible were it not for a free variable getting in the way. The grammar for annoying expressions is:

$$a \quad ::= \quad x \mid n \oplus a \mid a \oplus n \mid a \oplus a \mid a\,\overline{e}$$

Some expressions should be handled differently depending on context. If a constructor application appears in an empty context, there is not much we can do but to drive the argument expressions

$$
\begin{array}{lcll}
\mathcal{D}[\![n]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle n\rangle & \text{(R1)}\\
\mathcal{D}[\![x]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle x\rangle & \text{(R2)}\\
\mathcal{D}[\![g]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}_{app}(g,)_{\mathcal{R},\mathcal{G},\rho} & \text{(R3)}\\
\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho} & = & k\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho} & \text{(R4)}\\
\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle & \text{(R5)}\\
\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{G},\rho} & = & (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}) & \text{(R6)}\\
\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho}, \text{ where } n = n_1 + n_2 & \text{(R7)}\\
\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho} \oplus \mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}, \text{ if } e_1 \oplus e_2 = a & \text{(R8)}\\
 & & \mathcal{D}[\![e_2]\!]_{\mathcal{R}\langle e_1 \oplus []\rangle,\mathcal{G},\rho}, \text{ if } e_1 = n \text{ or } e_1 = a & \\
 & & \mathcal{D}[\![e_1]\!]_{\mathcal{R}\langle []\oplus e_2\rangle,\mathcal{G},\rho}, \text{ otherwise} & \\
\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & \text{(R9)}\\
\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let } \overline{x} = \overline{e} \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & \text{(R10)}\\
\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle [] \,\overline{e}\rangle,\mathcal{G},\rho} & \text{(R11)}\\
\mathcal{D}[\![\textbf{let } x = v \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle [v/x]f\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R12)}\\
\mathcal{D}[\![\textbf{let } x = y \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle [y/x]f\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R13)}\\
\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho}, \text{ if } x \in \textit{strict}(f) \text{ and f linear w.r.t } x & \text{(R14)}\\
 & & \textbf{let } x = \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho} \textbf{ in } \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}, \text{ otherwise} & \\
\mathcal{D}[\![\textbf{letrec } g = v \textbf{ in } e]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{[],\mathcal{G}',\rho}, \text{ where } \mathcal{G}' = \mathcal{G} \cup (g,v) & \text{(R15)}\\
\mathcal{D}[\![\textbf{case } x \textbf{ of } \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{case } x \textbf{ of } \{p_i \to \mathcal{D}[\![\mathcal{R}\langle [p_i/x]e_i\rangle]\!]_{[],\mathcal{G},\rho}\} & \text{(R16)}\\
\mathcal{D}[\![\textbf{case } k_j\,\overline{e} \textbf{ of } \{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j]\!]_{\mathcal{R},\mathcal{G},\rho} & \text{(R17)}\\
\mathcal{D}[\![\textbf{case } n_j \textbf{ of } \{n_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R18)}\\
\mathcal{D}[\![\textbf{case } a \textbf{ of } \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{case } \mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho} \textbf{ of } \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\} & \text{(R19)}\\
\mathcal{D}[\![\textbf{case } e \textbf{ of } \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle \textbf{case } [] \textbf{ of } \{p_i \to e_i\}\rangle,\mathcal{G},\rho} & \text{(R20)}
\end{array}
$$

Figure 5. Driving algorithm

(rule R4). On the other hand - if the application occurs at the head of a case expression, we may choose a branch on basis of the constructor and leave the arguments unevaluated in the hope of finding fold opportunities further down the syntax tree (rule R17).

The argumentation is analogous for lambda abstractions: if there is a surrounding context we perform a beta reduction, otherwise we drive its body.

Notice that the primitive operations ranged over by $\oplus$ can not be unfolded and transformed like ordinary functions can. If the arguments of a primitive operation are annoying our algorithm will simply leave the primitive operation in place (rule R8).

If we had a perfect strictness analysis and could decide whether an arbitrary expression will terminate or not, the only difference in results between our algorithm and a call-by-name counterpart would be for the non-terminating cases. In practice, we have to settle for an approximation, such as the simple analysis defined in Figure 6. One may speculate whether the transformations thus missed will have adverse effects on the usefulness of our algorithm in practice. We believe we have seen clear indications that this is not the case, and that crucial factor instead is the ability to inline function bodies irrespective of whether arguments are values or not.

Our algorithm always inlines functions unless the algorithm detects a risk of non-termination. Avoiding to inline an expression that could be inlined will give semantically equivalent, but syntactically different output from our algorithm. When the two programs are executed on a modern processor they will also most likely perform differently. Supero (Mitchell and Runciman 2008, Sec. 3.2) has a more advanced inlining strategy, something we leave for future work to investigate.

4.1 Application Rule

In the driving algorithm rule R3 and rule R9 refer to $\mathcal{D}_{app}(\,)$, defined in Figure 7. $\mathcal{D}_{app}(\,)$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation.

Figure 7 contains some new notation: we use $\equiv$ to denote equality of two expressions up to renaming of variables and $==$ for syntactical equivalence of expressions.

$$
\begin{array}{lcl}
GEN(e_1,e_2) & = & (\theta t, used \cup (\bigcup used'))\\
\multicolumn{3}{l}{\quad \text{where } (t_g,\theta_1',\theta_2') = msg(e_1,e_2),\ (t',\sigma) = split\, e_1}\\
\multicolumn{3}{l}{\qquad used' = \{u \,|(\_,u) \in S\}, \theta = \{e \,|(e,\_) \in S\}}\\
(t, used) & = & \mathcal{D}[\![t_g]\!]_{[],\mathcal{G},\rho}, \text{ if } t_g \neq x\\
 & & \mathcal{D}[\![t']\!]_{[],\mathcal{G},\rho}, \text{ otherwise}\\
S & = & \mathcal{D}[\![\theta_1']\!]_{[],\mathcal{G},\rho}, \text{ if } t_g \neq x\\
 & & \mathcal{D}[\![\sigma]\!]_{[],\mathcal{G},\rho}, \text{ otherwise}
\end{array}
$$

Figure 8. Generalization

Care needs to be taken to ensure that recursive functions are not inlined forever. The driving algorithm keeps a record of previously seen applications in the memoization list ρ; whenever it detects an expression that is equivalent (up to renaming of variables) to a previous expression, the algorithm creates a new recursive function h_n for some n. Whenever such an expression is encountered again, a call to h_n is inserted. This is not sufficient to guarantee termination of the algorithm, but the mechanism is crucial for the complexity improvements mentioned in Section 2.

To ensure termination, we use the homeomorphic embedding relation $\trianglelefteq$ to define a predicate called "the whistle". When the predicate holds for an expression we say that the whistle blows on that expression. The intuition is that when $e \trianglelefteq f, f$ contains all subexpressions of *e*, possibly embedded in other expressions. For any infinite sequence $e_0, e_1, \ldots$ there exists *i* and *j* such that $i < j$ and $e_i \trianglelefteq e_j$. This condition is sufficient to ensure termination.

In order to define the homeomorphic embedding we need a definition of uniform terms analogous to the work by Sørensen and Glück (1995), which we adjust slightly to fit our language.

Definition 4.1 (Uniform terms)**.** *Let s range over the set* $N \cup X \cup K \cup \{\mathbf{caseof}, \mathbf{let}, \mathbf{letrec}, \mathbf{primop}, \mathbf{lambda}, \mathbf{apply}\}$*, and let* $\mathbf{caseof}(\overline{e}), \mathbf{let}(\overline{e}), \mathbf{letrec}(\overline{v}, e), \mathbf{primop}(\overline{e}), \mathbf{lambda}(e)$*, and* $\mathbf{apply}(\overline{e})$ *denote a case, let, recursive let, primitive operation, lambda abstraction or application for all subexpressions* $\overline{e}, e$ *and* $\overline{v}$*. The set of terms T is the smallest set of arity respecting symbol applications* $s(\overline{e})$*.*

$$
\begin{array}{lcll}
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & (h'\,\overline{x}, \{(h', Nothing)\}), & \text{if } \exists (h',t) \in \rho\,.\, t \equiv \mathcal{R}\langle g\,\overline{e}\rangle \\
\multicolumn{4}{l}{\quad \text{where } \overline{x} = fv(\mathcal{R}\langle g\,\overline{e}\rangle)} \\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & (x', \{(h', Just\,\mathcal{R}\langle g\,\overline{e}\rangle)\}), & \text{if } \exists (h',t) \in \rho\,.\, t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \\
 & & & \text{and } t == \theta(\mathcal{R}\langle g\,\overline{e}\rangle) \\
\multicolumn{4}{l}{\quad \text{where } x' \text{ fresh}} \\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & GEN(\mathcal{R}\langle g\,\overline{e}\rangle, t) & \text{if } \exists (h',t) \in \rho\,.\, t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & GEN(\mathcal{R}\langle g\,\overline{e}\rangle, head(W)) & \text{if W} \neq \emptyset \\
 & & (\textbf{letrec}\, h = \lambda\overline{x}.e' \,\textbf{in}\, h\,\overline{x}, used), & \text{if found} \neq \emptyset \\
 & & (e', used), & \text{otherwise} \\
\multicolumn{4}{l}{\quad \text{where } (g = v) \in \mathcal{G}, (e', used) = \mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}} \\
\multicolumn{4}{l}{\qquad \overline{x} = fv(\mathcal{R}\langle g\,\overline{e}\rangle), \rho' = \rho \cup (h, \mathcal{R}\langle g\,\overline{e}\rangle), h \text{ fresh},} \\
\multicolumn{4}{l}{\qquad W = \{e | (n, Just\,e) \in used, n == h\}} \\
\multicolumn{4}{l}{\qquad found = \{n | (n, Nothing) \in used, n == h\}}
\end{array}
$$

Figure 7. Driving of applications

Definition 4.2 (Homeomorphic embedding)**.** *Define* $\trianglelefteq$ *as the smallest relation on* T *satisfying:*

$$x \trianglelefteq y, \;\; n_1 \trianglelefteq n_2, \;\; \frac{e \trianglelefteq f_i \text{ for some } i}{e \trianglelefteq s(f_1, \ldots, f_n)},$$

$$\frac{e_1 \trianglelefteq f_1, \ldots, e_n \trianglelefteq f_n}{s(e_1, \ldots, e_n) \trianglelefteq s(f_1, \ldots, f_n)}$$

Examples of the homeomorphic embedding are shown in Figure 9.

Whenever the whistle blows, our algorithm splits the input expression into strictly smaller terms that are driven separately in the empty context. This might expose new folding opportunities, and allows the algorithm to remove intermediate structures in subexpressions. The design follows the positive supercompilation as outlined by Sørensen (2000), except that we need to substitute the transformed expressions back instead of pulling them out into let-statements, in order to preserve strictness. Our algorithm is also more complicated because we perform the program extraction immediately instead of constructing a large tree and extracting the program in a separate pass.

Splitting expressions is rather intricate, and two mechanisms are needed, the first is the most specific generalization that entails the smallest possible loss of knowledge, and is defined as:

Definition 4.3 (Most specific generalization)**.**

- *An* instance *of a term* e *is a term of the form* θe *for some substitution* θ*.*
- *A* generalization *of two terms* e *and* f *is a triple* $(t_g, \theta_1, \theta_2)$*, where* θ_1, θ_2 *are substitutions such that* $\theta_1 t_g \equiv e$ *and* $\theta_2 t_g \equiv f$*.*
- *A* most specific generalization *(msg) of two terms* e *and* f *is a generalization* $(t_g, \theta_1, \theta_2)$ *such that for every other generalization* $(t'_g, \theta'_1, \theta'_2)$ *of* e *and* f *it holds that* t_g *is an instance of* t'_g*.*

We refer to t_g as the ground term. For background information and an algorithm to compute most specific generalizations, see Lassez et al. (1988). Figure 9 also contains examples of the msg.

The most specific generalization is not always sufficient to split expressions. For some expressions it will return the ground term as a variable, and the respective θs equal to the input terms. If this happens, a function split is needed, defined as:

Definition 4.4 (Split)**.** *For* $t \in T$ *we define* $split(t)$ *by:*

$$split(s(e_1, \ldots, e_n)) = (s(x_1, \ldots, x_n), [e_1/x_1, \ldots, e_n/x_n])$$

with $x_1, \ldots, x_n$ *fresh.*

Alternative 2 of $\mathcal{D}_{app}(\,)$ is for upwards generalization, and alternatives 3 and 4 are for the downwards generalization. This is exemplified below. The generalization algorithm is shown in Figure 8. If the ground term t_g is a variable the algorithm drives the output from split, otherwise it will drive the output from the msg.

All the examples of how our algorithm works in Section 2 eventually terminate through a combination of alternative 1 and alternative 4 (found $\neq \emptyset$) of $\mathcal{D}_{app}(\,)$.

The second alternative of $\mathcal{D}_{app}(\,)$ in combination with the fourth alternative ($W \neq \emptyset$) is useful when transforming function calls that have the same parameter appearing twice, for example *append xs xs* as shown in Figure 10. For space reasons we have omitted several intermediate steps that do not contribute to the understanding of the current discussion.

The third alternative is used when terms are "growing" in some sense. An example of *reverse* with an accumulating parameter is shown in Figure 11 with the definition of reverse as:

$$
\begin{array}{l}
rev\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [] \ \rightarrow\ ys \\
\qquad\qquad (x' : xs') \ \rightarrow\ rev\ xs'\ (x' : ys)
\end{array}
$$

$\mathcal{D}[\![\ append\ xs\ xs]\!]$ (*)

(Put (h_0, *append xs xs*) in ρ and transform according to the rules of the algorithm)

$$
\begin{array}{l}
\textbf{case}\ xs\ \textbf{of} \\
\quad [] \ \rightarrow\ xs \\
\quad (x' : xs') \ \rightarrow\ \mathcal{D}[\![\ x']\!] \ : \mathcal{D}[\![\ append\ xs'\ xs]\!]
\end{array}
$$

(Focus on $\mathcal{D}[\![\ append\ xs'\ xs]\!]$ and recall that ρ contains *append xs xs* so alternative 2 of $\mathcal{D}_{app}(\,)$ is triggered and the transformation returns (x, $\{(h_0, Just\ (append\ xs'\ xs))\}$). This returns all the way up to (*) and restarts the transformation there with $W = \{append\ xs'\ xs\}$)

$\mathcal{D}[\![\ append\ xs\ xs]\!]$

= (Generalize the expression with $append\ xs'\ xs$)

$[\mathcal{D}[\![\ xs']\!]/x,\ \mathcal{D}[\![\ xs']\!]/y]\ \mathcal{D}[\![\ append\ x\ y]\!]$

$$
\begin{array}{ll}
= & \textbf{letrec}\ h_0\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of} \\
 & \qquad\qquad\qquad [] \ \rightarrow\ ys \\
 & \qquad\qquad\qquad (x' : xs') \ \rightarrow\ x' : h_0\ xs'\ ys \\
 & \textbf{in}\ h_0\ xs\ xs
\end{array}
$$

Figure 10. Example of upwards generalization

e		f	t_g	θ_1	θ_2
e	$\trianglelefteq$	$Just\ e$	x	$[e/x]$	$[Just\ e/x]$
$Right\ e$	$\trianglelefteq$	$Right\ (e,\ e')$	$Right\ x$	$[e/x]$	$[(e, e')/x]$
$fac\ y$	$\trianglelefteq$	$fac\ (y\ -\ 1)$	$fac\ x$	$[y/x]$	$[(y-1)/x]$

Figure 9. Examples of the homeomorphic embedding and the msg

$\mathcal{D}[\![\ rev\ xs\ []\ ]\!]$

(Put (h_0, $rev\ xs\ []$) in ρ and transform the program according to the rules of the algorithm)

$$\begin{array}{l}\textbf{case}\ xs\ \textbf{of}\\ \quad [] \rightarrow []\\ \quad (x' : xs') \rightarrow \mathcal{D}[\![\ rev\ xs'\ (x' : [])]\!]\end{array}$$

(Focus on the second branch and recall that ρ contains $rev\ xs\ []$ so alternative 3 of $\mathcal{D}_{app}(\ )$ is triggered and the expression is generalized)

$\mathcal{D}[\![\ rev\ xs'\ (x' : [])]\!]$

(Generalize the expression with $rev\ xs\ []$)

$[\mathcal{D}[\![\ (x' : [])]\!]/zs]\mathcal{D}[\![\ rev\ xs'\ zs]\!]$

(Put (h_1, $rev\ xs'\ zs$) in ρ and transform according to the rules of the algorithm)

$$\begin{array}{ll}= & \textbf{letrec}\ h_1\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of}\\ & \qquad\qquad\qquad\qquad [] \rightarrow ys\\ & \qquad\qquad\qquad\qquad (x' : xs') \rightarrow h_1\ xs'\ (x' : ys)\\ & \textbf{in}\ h_1\ xs'\ (x' : [])\end{array}$$

(Putting the two parts together)

$$\begin{array}{l}\textbf{case}\ xs\ \textbf{of}\\ \quad [] \rightarrow []\\ \quad (x' : xs') \rightarrow\\ \quad\quad \textbf{letrec}\ h_1\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of}\\ \quad\quad\quad\quad\quad\quad\quad\quad [] \rightarrow ys\\ \quad\quad\quad\quad\quad\quad\quad\quad (x' : xs') \rightarrow h_1\ xs'\ (x' : ys)\\ \quad\quad \textbf{in}\ h_1\ xs'\ (x' : [])\end{array}$$

Figure 11. Example of downwards generalization

To actually perform the generalization both upwards and downwards the driving algorithm must propagate information in both directions, hence the return type of $\mathcal{D}[\![\ ]\!]$ must be changed to a pair, which in turn requires glue code for all the rules in Figure 5. This glue code simply merges the second component of the return values. The following is an example of a Haskell implementation of rule R4 with the return type changed to a pair and the return values from the subcomputations are merged and propagated upwards:

$$\begin{array}{l}drive\ r\ rho\ (ECon\ name\ args)\ =\\ \quad (ECon\ name\ args',\ concat\ used)\\ \quad \textbf{where}\\ \quad\quad (args',\ used)\ =\ unzip\ (map\ dr\ args)\\ \quad\quad dr\ =\ drive\ emptyContext\ rho\end{array}$$

5. Correctness

The problem with previous deforestation and supercompilation algorithms in a call-by-value context is that they might change termination properties of programs. We prove that our supercompiler both terminates and does not alter whether a program terminates or not. The complete proofs are available from a companion technical report (Jonsson and Nordlander 2008), but they do not reveal anything unexpected.

5.1 Termination

In order to prove that the algorithm terminates we show that each recursive application of $\mathcal{D}[\![\ ]\!]$ in the right-hand sides of Figure 5 and 7 has a strictly smaller weight than the left-hand side. The weight of an expression is one plus the sum of the weight of its subexpressions, where variables, primitive numbers and function names have weight two. The weight of a fresh variable not in the initial input is one.

Definition 5.1. *The weight of a variable x in the initial input, a primitive number n, and a function name g is 2. The weight of a fresh variable not in the initial input is 1. The weight of any composite expression ($n \geq 1$) is $|s(e_1, \ldots, e_n)| = 1 + \sum_{i=1}^{n} |e_i|$.*

Definition 5.2. *Let S be a set with a relation $\leq$. Then $(S, \leq)$ is a quasi-order if $\leq$ is reflexive and transitive.*

Definition 5.3. *Let $(S, \leq)$ be a quasi-order. $(S, \leq)$ is a well-quasi-order if, for every infinite sequence $s_0, s_1, \ldots \in S$, there are $i < j$ with $s_i \leq s_j$*

The weight of the entire transformation is a triple that contains the maximum length of the memoization list ρ denoted by *N*, the weight of the term being transformed and the weight of the current term in focus. That such an *N* exists follows from the homeomorphic embedding is a well-quasi-order and Kruskal's Tree Theorem (Dershowitz 1987):

Theorem 5.4 (Kruskal's Tree Theorem)**.** *If S is a finite set of function symbols, then any infinite sequence $t_1, t_2, \ldots$ of terms from the set S contains two terms t_i and t_j with $i < j$ such that $t_i \trianglelefteq t_j$.*

We need to show that the memoization list ρ only contains elements that were in the initial input program:

Lemma 5.5. *The second component of the memoization list, ρ, can only contain terms from the set T.*

Definition 5.6. *The weight of a call to the driving algorithm is $|\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle e\rangle|, |e|)$*

Tuples must be ordered for us to tell whether the weight of a term actually decreases from driving it. We use the standard lexical order between tuples.

With these definitions in place, we can formulate a lemma that the weight is decreasing in each step of our algorithm.

Lemma 5.7. *For each rule R: $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho} = e_1$ in Figure 5 and Figure 7 and each recursive application $\mathcal{D}[\![e']\!]_{\mathcal{R}',\mathcal{G},\rho'}$ in e_1, $|\mathcal{D}[\![e']\!]_{\mathcal{R}',\mathcal{G},\rho'}| < |\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}|$*

Lemma 5.8 (Totality). *For all well-typed expressions e, $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}$ is matched by a unique rule in Figure 5.*

Proposition 5.9 (Termination). *The driving algorithm $\mathcal{D}[\![\ ]\!]$ terminates for all well-typed inputs.*

Proof. The weight of the transformation is defined because the homeomorphic embedding is a well-quasi-order combined with Kruskal's Tree Theorem. Lemma 5.5 guarantees that the memoization list ρ only contains terms from the initial input. By Lemma 5.7 the weight of the transformation decreases for each step and by Lemma 5.8 we know that each recursive application will match a rule.

Since $<$ is well-founded over triples of natural numbers the system will eventually terminate. □

5.2 Total Correctness

We define the standard notions of operational approximation and equivalence. A general context C which has zero or more holes in the place of some subexpressions is introduced.

Definition 5.10 (Operational Approximation and Equivalence).

- *e operationally approximates e', $e \sqsubseteq e'$, if for all contexts C such that C[e], C[e'] are closed, if evaluation of C[e] terminates then so does evaluation of C[e'].*
- *e is operationally equivalent to e', $e \cong e'$, if $e \sqsubseteq e'$ and $e' \sqsubseteq e$*

The correctness of deforestation in a call-by-name setting has previously been shown by Sands (1996) using his improvement theory. Notice that improvement $\trianglerighteq$ below is not the same as the homeomorphic embedding $\trianglelefteq$ defined previously. We use Sands's definitions for improvement and strong improvement:

Definition 5.11 (Improvement, Strong Improvement).

- *e is improved by e', $e \trianglerighteq e'$, if for all contexts C such that C[e], C[e'] are closed, if computation of C[e] terminates using n function calls, then computation of C[e'] also terminates, and uses no more than n function calls.*
- *e is strongly improved by e', $e \trianglerighteq_s e'$, iff $e \trianglerighteq e'$ and $e \cong e'$.*

We use $e \mapsto^k v$ to denote that *e* evaluates to *v* using *k* function calls, and any other reduction rule as many times as it needs, and $e' \mapsto^{\leq k} v'$ to denote that *e'* evaluates to *v'* with at most *k* function calls and any other reduction rule as many times as it needs.

Definition 5.12 (Cost equivalence). *The expressions e and e' are cost equivalent, $e \trianglelefteq\trianglerighteq e'$ iff $e \trianglerighteq e'$ and $e' \trianglerighteq e$*

Cost equivalence implies strong improvement. If two terms evaluate with the same cost to two cost equivalent expressions, then the initial terms are also cost equivalent:

Lemma 5.13 (Sands (1996)). *If $e_1 \mapsto^r e_1'$ and $e_2 \mapsto^r e_2'$ then $(e_1' \trianglelefteq\trianglerighteq e_2' \Leftrightarrow e_1 \trianglelefteq\trianglerighteq e_2)$.*

With these definitions in place, total correctness for a transformation can be stated:

Theorem 5.14 (Sands). *If $e \trianglerighteq_s e'$, a transformation that replaces e by e' is totally correct.*

Improvement theory in a call-by-value setting requires Sands's operational metatheory for functional languages (Sands 1997).

Proposition 5.15 (Total Correctness). *Let e be an expression, and ρ an environment such that*

- *the range of ρ contains only closed expressions, and*
- *$fv(e) \cap dom(\rho) = \emptyset$*

then $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})$.

Proof. We sketch on the proof for the first half of rule R14. All the remaining rules have similar proofs except for rule R9, where the proof is similar in structure to the proof by Sands (1996, p. 24). We have that $\rho(\mathcal{D}[\![\mathbf{let}\, x = e\, \mathbf{in}\, f]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho})$. Evaluating the input term will eventually yield a context $\mathcal{E}\langle\rangle$ with a term: $\mathcal{R}\langle \mathbf{let}\, x = e\, \mathbf{in}\, f\rangle \mapsto^r \mathcal{R}\langle \mathbf{let}\, x = v\, \mathbf{in}\, f\rangle \mapsto \mathcal{R}\langle [v/x]f\rangle \mapsto^s \mathcal{E}\langle v\rangle$, and evaluating the input to the recursive call yields (remember that f is linear w.r.t x): $\mathcal{R}\langle [e/x]f\rangle \mapsto^s \mathcal{E}\langle e\rangle \mapsto^r \mathcal{E}\langle v\rangle$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 5.13 their ancestor terms are cost equivalent, $\mathcal{R}\langle \mathbf{let}\, x = e\, \mathbf{in}\, f\rangle \trianglelefteq\trianglerighteq \mathcal{R}\langle [e/x]f\rangle$, and cost equivalence implies strong improvement. By the induction hypothesis $\mathcal{R}\langle [e/x]f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho})$, and therefore $\mathcal{R}\langle \mathbf{let}\, x = e\, \mathbf{in}\, f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\, x = e\, \mathbf{in}\, f]\!]_{\mathcal{R},\mathcal{G},\rho})$. □

6. Benchmarks

We provide measurements from a set of common examples from the literature on deforestation. We show that our positive supercompiler does remove intermediate structures, and can improve the performance by an order of magnitude for certain benchmarks. We have left out the full details of the instrumentation of the runtime system and the transformed result of each benchmark for space reasons, but they are available in a separate report (Jonsson 2008).

All measurements were performed on an idle machine running in an xterm. Each test was run 10 consecutive times and the best result was selected. The best result was selected since it must appear under the minimum of other activity of the operating system. The number of allocations and total allocation size remains constant over all runs.

The raw data for the time, size and allocation measurements are shown in Table 1. The time column is number of clockticks from the RDTSC instruction available in Intel/AMD processors, and the binary size is in bytes as reported by *ls*. The total number of allocations and the total memory size allocated by the program are displayed in each column.

The binary sizes are slightly increased by the supercompiler, but the runtimes are all faster. The main reason for the performance improvement is the removal of intermediate structures, reducing the amount of memory allocations.

The work on Supero by Mitchell and Runciman (2008) shows that there are open problems for supercompiling large Haskell programs. These problems are mainly relating to speed, both of the compiler, and of the transformed program. When they profiled Supero, they found that the majority of the time was spent in the homeomorphic embedding test. Our algorithm performs the test on a smaller part of the tree, so there is reason to believe that less time will be spent in the test for our algorithm. The complexity of the homeomorphic embedding has been investigated separately by Narendran and Stillman (1987) and they give an algorithm of complexity $O(size(e) \times size(f))$ to decide whether $e \trianglelefteq f$. We expect the same problems that Mitchell and Runciman observed to appear in a call-by-value context as well, and intend to investigate them now that we have a theoretical foundation for our algorithm.

6.1 Double Append

As previously seen, appending three lists saves one traversal over the first list. This is an example by Wadler (1990), and the intermediate structure is fused away by our supercompiler. Three strings of 9000 characters each were appended to each other into a 27 000 characters long string. The number of allocations goes down, and one iteration over the first string is avoided. The binary size increases 1316 bytes, on a binary of roughly 90k.

	Time		Binary size		Allocations		Alloc Size	
Benchmark	Before	After	Before	After	Before	After	Before	After
Double Append	105 844 704	89 820 912	89 484	90 800	270 035	180 032	2 160 280	1 440 256
Factorial	21 552	21 024	88 968	88 968	9	9	68	68
Flip a Tree	2 131 188	237 168	95 452	104 704	20 504	57	180 480	620
Sum of Squares of a Tree	276 102 012	28 737 648	95 452	104 912	4 194 338	91	29 360 496	908
Kort's Raytracer	12 050 880	7 969 224	91 968	91 460	60 021	17	320 144	124

Table 1. Time, space and allocation measurements

6.2 Factorial

There are no intermediate lists created in a standard implementation, so any performance improvements come from inlining or reductions. One recursion and a couple of reductions are eliminated, thereby slightly reducing the runtime. The allocations remain the same and the final binary size remains unchanged.

6.3 Flip a Tree

Flipping a tree is another example by Wadler (1990), and just like Wadler we perform a double flip (thus restoring the original tree) before printing the total sum of all leaves. The supercompiler manages to eliminate the double flip. The total number of allocations and the total size of allocations is reduced. The runtime also goes down by an order of magnitude! The binary size increases close to 10% however.

6.4 Sum of Squares of a Tree

Computing the sum of the squares of the data members of a tree is the final example by Wadler (1990). Almost all allocations are removed by our supercompiler, but the binary size is increased by nearly 10%. The runtime is improved by an order of magnitude.

6.5 Kort's Raytracer

The inner loop of a Raytracer (Kort 1996) is extracted and transformed. The total runtime, the number of allocations, the total size of allocations and the binary size are all decreased.

7. Extensions

There are several ways the driving algorithm can be extended to make it more powerful. We show that with an extended Let-rule in combination with a disabled whistle for closed expressions we can evaluate many closed expressions.

If the let-expression contains no free variables we can drive either the right hand side or the body and see what the result is. Another option is to augment rule R14 with a second and third alternative as shown in Figure 12.

The reasoning behind this is that a closed expression contains all information needed to evaluate it, thus a fold should be unnecessary. If the expression diverges, then so does the final program at that point.

The immediate question following from the above is whether this is beneficial for expressions that are not closed, a question we have no definite answer to. An example of the benefit of driving the body is $bodyEx$ as shown below.

$$
\begin{array}{ll}
& bodyEx = \textbf{let}\ x\ =\ e\ \textbf{in case}\ []\ \textbf{of}\ \{[]\ \rightarrow\ x; (x : xs)\ \rightarrow\ y\} \\
& \quad \mathcal{D}[\![\ bodyEx\ ]\!] \\
= & \quad \mathcal{D}[\![\ \textbf{let}\ x\ =\ e\ \textbf{in case}\ []\ \textbf{of}\ \{\ []\ \rightarrow\ x; (x : xs)\ \rightarrow\ y\ \}]\!] \\
= & \quad \mathcal{D}[\![\ e\ ]\!]
\end{array}
$$

We can see how the body becomes strict after driving it, which opens up for further transformations.

Duplicating code to enable further transformations might sometimes be beneficial. Removing the linearity constraint of rule R14 could enable further transformations, but it could just as well turn out to have duplicated work, forcing multiple evaluations of the "same" expression, as well as growth in code size. The current algorithm has no means of finding a suitable trade-off. Obtaining a more refined behavior in this respect is left for future work.

8. Related Work

There exists much literature concerning algorithms that remove intermediate structures in functional programs. However, most of it is in a call-by-name or call-by-need context which makes it a different, yet difficult, problem. We therefore start our survey of related work with one call-by-value transformation, and then look at the related transformations from call-by-name and call-by-need contexts.

8.1 Lightweight Fusion

Lightweight Fusion (Ohori and Sasano 2007) works by promoting a function through the fix point operator and guarantees termination by limiting each function to be inlined at most once. They implement the transformation in a variant of a compiler for Standard ML and present some benchmarks. The algorithm is proven correct for a call-by-name language. It is explicitly mentioned that their goal is to extend the transformation to work for an impure call-by-value functional language.

Comparing lightweight fusion to our positive supercompiler is somewhat difficult, the algorithms are not very similar in themselves. Comparing results of the algorithms is more straightforward – the restriction to only inline functions once makes lightweight fusion unable to handle successive applications of the same function and mutually recursive functions, something the positive supercompiler handles gracefully.

Considering the early stage of their work, we still find it an interesting approach that seems to solve a lot of problems.

8.2 Deforestation

Deforestation, removing intermediate structures from programs, was pioneered by Wadler (1990) for a first order language more than fifteen years ago. The initial deforestation had support for higher order macros, incapable of fully emulating higher order functions.

Marlow and Wadler (1992) addressed the restriction to a first-order language when they presented a deforestation algorithm for a higher order language. This work was refined in Marlow's (1995) dissertation, where he also related deforestation to the cut-elimination principle of logic. Chin (1994) has also generalised Wadler's deforestation to higher-order functional programs by using syntactic properties to decide which terms that can be fused.

Both Hamilton (1996) and Marlow (1995) have proven that their deforestation algorithms terminate. More recent work by Hamilton (2006) extends deforestation to handle a wider range of functions, with an easy to recognise treeless form, giving more transparency for the programmer.

Alimarine and Smetsers (2005) have improved the producer and consumer analyses in Chin's (1994) algorithm to be based on

$$\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} = \begin{array}{ll} \mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho}, & \text{if } x \in \textit{strict}(f) \text{ and f linear w.r.t x} \\ \mathcal{D}[\![\mathcal{R}\langle [v/x]f\rangle]\!]_{[],\mathcal{G},\rho}, & \text{if } \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho} = v \\ \mathcal{D}[\![\mathcal{R}\langle [e/x]f'\rangle]\!]_{[],\mathcal{G},\rho}, & \text{if } \mathcal{D}[\![f]\!]_{[],\mathcal{G},\rho} = f',\ x \in \textit{strict}(f') \text{ and f' linear w.r.t x} \\ \textbf{let } x = \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho} \textbf{ in } \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}, & \text{otherwise} \end{array}$$

Figure 12. Extended Let-rule

semantics rather than syntax. They show their algorithm can remove much of the overhead introduced from generic programming (Hinze 2000).

While all this work is algorithmically rather close to ours due to the close relation between deforestation and positive supercompilation, it is in a call-by-name or call-by-need context.

8.3 Supercompilation

Closely related to deforestation is *supercompilation* (Turchin 1979, 1980, 1986a,b). Supercompilation both removes intermediate structures, achieves partial evaluation as well as some other optimisations. In partial evaluation terminology, the decision of when to inline is taken online. The initial studies on supercompilation were for the functional language Refal (Turchin 1989).

The *positive supercompiler* (Sørensen et al. 1996) is a variant which only propagates positive information, such as equalities. The propagation is done by unification and the work highlights how similar deforestation and positive supercompilation really are. Narrowing (Albert and Vidal 2001) is the functional logic programming community equivalent of positive supercompilation but formulated as a term rewriting system. They also deal with nondeterminism from backtracking, which makes the algorithm more complicated.

Strengthening the information propagation mechanism to propagate not only positive, but also negative information, yields *perfect supercompilation* (Secher 1999; Secher and Sørensen 2000). Negative information is the opposite of positive information, inequalities. These inequalities can be used to prune branches that are certainly not taken in case-statements for example.

More recently, Mitchell and Runciman (2008) have worked on supercompiling Haskell. They report runtime reductions of up to 55% when their supercompiler is used in conjunction with GHC.

The positive supercompiler by Sørensen et al. (1996) is the immediate ancestor of our work, but we extended it to a higher-order language and converted it to work on call-by-value languages.

8.4 Generalized Partial Computation

GPC (Futamura and Nogi 1988) uses a theorem prover to extract additional properties about the program being specialized. Among these properties are the logical structure of a program, axioms for abstract data types, and algebraic properties of primitive functions. Early work on GPC was performed by Takano (1991).

A theorem prover is used on top of the transformation and whenever a test is encountered the theorem prover verifies whether one or more branches can be taken. Information about the predicate which was tested is propagated along the branches that are left in the resulting program. The reason GPC is such a powerful transformation is because it assumes the unlimited power of a theorem prover.

Futamura et al. (2002) has applied GPC in a call-by-value setting in a system called WSDFU (Waseda Simplify-Distribute-Fold-Unfold), reporting many successful experiments where optimal or near optimal residual programs are produced. It is unclear whether WSDFU preserves termination behaviour or if it is a call-by-name transformation applied to a call-by-value language.

We note that the rules for the first order language presented by Takano (1991) are very similar to the positive supercompiler, but the theorem prover required might exclude the technique as a candidate for automatic compiler optimisations. The lack of termination guarantees for the transformation might be another obstacle. Considering the similarity of GPC and positive supercompilation it should be straight forward to convert GPC to work on a call-by-value language, which makes it rather close to our work.

8.5 Other Transformations

Considering the vast amount of research conducted on program transformations, we only briefly survey other related transformations.

8.5.1 Partial Evaluation

Partial evaluation (Jones et al. 1993) is another instance of Burstall and Darlington's (1977) informal class of fold/unfold transformations.

If the partial evaluation is performed offline, the process is guided by program annotations that tells when to fold, unfold, instantiate and define. Binding-Time Analysis (BTA) is a a program analysis that annotates operations in the input program based on whether they are statically known or not.

Partial evaluation does not remove intermediate structures, something we deem necessary to enable the programmer to write programs in the clear and concise listful style. Both deforestation and supercompilation simulate call-by-name evaluation in the transformer, whereas partial evaluation simulates call-by-value. It is suggested by Sørensen et al. (1994) that this might affect the strength of the transformation.

8.5.2 Short Cut Deforestation

Short cut deforestation (Gill et al. 1993; Gill 1996) takes a different approach to deforestation, sacrificing some generality by only working on lists.

The idea is that the constructors *Nil* and *Cons* can be replaced by a *foldr* consumer, and a special function *build* is used for the transformation to recognize the producer and enforce the type requirement. Lists using *build/foldr* can easily be removed with the *foldr/build* rule: $\mathit{foldr}\ f\ c\ (\mathit{build}\ g) = g\ f\ c$.

This shifts the burden from the compiler on to the programmer or compiler writer to make sure list-traversing functions are written using *build* and *foldr*, thereby cluttering the code with information for the optimiser and making it harder to read and understand for humans.

Gill implemented and measured short cut deforestation in GHC using the nofib benchmark suite (Partain 1992). Around a dozen benchmarks improved by more than 5%, average was 3% and only one example got noticeably worse, by 1%. Heap allocations were reduced, down to half in one particular case.

The main argument for short cut deforestation is its simplicity on the compiler side compared to full-blown deforestation. GHC as of today contains a variant of the short cut deforestation implemented by use of the rewrite rules (Jones et al. 2001) available in GHC.

Ghani and Johann (2008) have generalized the *foldr/build* rule to a *fold/superbuild* rule that can eliminate intermediate structures of inductive types without disturbing the contexts in which they are situated.

8.5.3 Type-inference Based Short Cut Deforestation

Type-inference can be used to transform the producer of lists into the abstracted form required by short cut deforestation, and this is exactly what Chitil (2000) does. Given a type-inference algorithm which infers the most general type, Chitil is able to determine the list constructors that need to be replaced in one pass.

From the principal type property of the type inference algorithm Chitil was able to deduce completeness of the list abstraction algorithm. This completeness guarantees that if a list can be abstracted from a producer by abstracting its list constructors, then the list abstraction algorithm will do so.

The implications of the completeness is that a foldr consumer can be fused with nearly any producer. A reason list constructors might not be abstractable from a producer is that they do not occur in the producer expression but in the definition of a function which is called by the producer. A worker/wrapper scheme proposed ensures that these list constructors are moved to the producer to make the list abstraction possible.

Chitil compared heap allocation and runtime between the short cut deforestation in GHC 4.06 and a program optimised with the type-inference based short cut deforestation. The example in question was the *n-queens* problem, where n was set to 10 in order to make I/O time less significant than a smaller instance would have. Heap allocation went from 33 to 22 megabytes and runtime from 0.57 seconds to 0.51 seconds.

The completeness property and the fact that the programmer does not have to write any special code in combination with the promising results from measurements suggests type-inference based short cut deforestation is a practical optimisation.

8.5.4 Zip Fusion

Takano and Meijer (1995) noted that the foldr/build rule for short cut deforestation had a dual. This is the *destroy/unfoldr* rule used in Zip Fusion (Svenningsson 2002) which has some interesting properties.

It can remove all argument lists from a function which consumes more than one list. The method described by Svenningsson will remove all intermediate lists in $zip\ [1..n]\ [1..n]$, one of the main criticisms against the *foldr/build* rule. The technique can also remove intermediate lists from functions which consume their lists using accumulating parameters, a known problematic case when fusing functions that most techniques can not handle. The *destroy/unfoldr* rule is defined as:

$$destroy\ g\ (unfoldr\ psi\ e)\ =\ g\ psi\ e$$

The method is simple, and can be implemented the same way as short cut deforestation. It still suffers from the drawback that the programmer or compiler writer has to make sure the list traversing functions are written using *destroy* and *unfoldr*.

9. Conclusions

A positive supercompiler, for a higher-order call-by-value language, that includes folding has been presented. We have proven it correct.

The adjustment to the algorithm for preserving call-by-value semantics is new and works surprisingly well for many examples that were intended to show the usefulness of call-by-name transformations.

9.1 Future Work

We believe that the linearity restriction of rule R14 in the proof of correctness is not necessary for the soundness of our algorithm, but have not found a way to prove it yet. We will investigate how the concept of an inline budget may be used to obtain good balance between code size and inlining benefits.

More work could be done on strictness analysis component of our supercompiler. We do not intend to focus on that subject, though; instead we hope that the modular dependency on strictness analysis will allow our supercompiler to readily take advantage of general improvements in the area.

Acknowledgments

The authors would like to thank Simon Marlow, Duncan Coutts and Neil Mitchell for valuable discussions. We would also like to thank Viktor Leijon and the anonymous referees for providing useful comments that helped improve the presentation and contents and Germán Vidal for explaining narrowing to us.

References

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

A. Alimarine and S. Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings*, volume 3350 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2005. ISBN 3-540-24362-3.

R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

W-N. Chin. Safe fusion of functional expressions II: Further improvements. *J. Funct. Program*, 4(4):515–555, 1994.

O. Chitil. *Type-Inference Based Deforestation of Functional Programs*. PhD thesis, RWTH Aachen, October 2000.

N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, 1987.

Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.

Y. Futamura, Z. Konishi, and R. Glück. Program transformation system based on generalized partial computation. *New Gen. Comput.*, 20(1): 75–99, 2002. ISSN 0288-3635.

N. Ghani and P. Johann. Short cut fusion of recursive programs with computational effects. In P. Achten, P. Koopman, and M. T. Morazán, editors, *Draft Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP)*, number ICIS–R08007, 2008.

A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 1993*, 1993.

A. J. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Univ. of Glasgow, January 1996.

G. W. Hamilton. Higher order deforestation. In *PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 213–227, London, UK, 1996. Springer-Verlag. ISBN 3-540-61756-6.

G. W. Hamilton. Higher order deforestation. *Fundam. Informaticae*, 69 (1-2):39–61, 2006.

R. Hinze. *Generic Programs and Proofs*. PhD thesis, Habilitationsschrift, Bonn University, 2000.

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

N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993. ISBN 0-13-020249-5.

S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW'2001), 2nd September 2001, Firenze, Italy.*, Electronic Notes in Theoretical

Computer Science, Vol 59. Utrecht University, September 28 2001. UU-CS-2001-23.

P. A. Jonsson. Positive supercompilation for a higher-order call-by-value language. Licentiate thesis, Luleå University of Technology, Sweden, Jun 2008.

P. A. Jonsson and J. Nordlander. Positive Supercompilation for a Higher Order Call-By-Value Language: Extended Proofs. Technical Report 2008:17, Department of Computer science and Electrical engineering, Luleå University of Technology, October 2008.

J. Kort. Deforestation of a raytracer. Master's thesis, University of Amsterdam, 1996.

J-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

X. Leroy. The Objective Caml system: Documentation and user's manual, 2008. With D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. Available from `http://caml.inria.fr` (1996–2008).

S. Marlow and P. Wadler. Deforestation for higher-order functions. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 154–165. Springer, 1992. ISBN 3-540-19820-2.

S. D. Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Department of Computing Science, University of Glasgow, April 27 1995.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

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.

P. Narendran and J. Stillman. On the Complexity of Homeomorphic Embeddings. Technical Report 87-8, Computer Science Department, State Univeristy of New York at Albany, March 1987.

J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The Timber home page, 2008. URL `http://www.timber-lang.org`.

A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

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

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

D. Sands. From SOS rules to proof principles: An operational metatheory for functional languages. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1997.

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

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.

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

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.

M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 485–500. Berlin: Springer-Verlag, 1994.

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

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP*, pages 124–132, 2002.

D. Syme. The F# programming language, Jun 2008. URL `http://research.microsoft.com/fsharp`.

A. Takano. Generalized partial computation for a lazy functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 1–11. New York: ACM, 1991.

A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *FPCA*, pages 306–313, 1995.

V.F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

V.F. Turchin. Semantic definitions in Refal and automatic production of compilers. In N.D. Jones, editor, *Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94)*, pages 441–474. Berlin: Springer-Verlag, 1980.

V.F. Turchin. Program transformation by supercompilation. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985 (Lecture Notes in Computer Science, vol. 217)*, pages 257–281. Berlin: Springer-Verlag, 1986a.

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

V.F. Turchin. *Refal-5, Programming Guide & Reference Manual*. Holyoke, MA: New England Publishing Co., 1989.

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