Strengthening Supercompilation For Call-By-Value Languages

Peter A. Jonsson and Johan Nordlander

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

Abstract. A termination preserving supercompiler for a call-by-value language sometimes fails to remove intermediate structures that a supercompiler for a call-by-name language would remove. This discrepancy in power stems from the fact that many function bodies are either non-linear in use of an important variable or often start with a pattern match on their first argument and are therefore not strict in all their arguments. As a consequence, intermediate structures are left in the output program, making it slower. We present a revised supercompilation algorithm for a call-by-value language that propagates let-bindings into case-branches and uses termination analysis to remove dead code. This allows the algorithm to remove all intermediate structures for common examples where previous algorithms for call-by-value languages had to leave the intermediate structures in place.

1 Introduction

Intermediate lists in functional programs allows the programmer to write clear and concise programs, but carry a run time cost since list cells need to be both allocated and garbage collected. Much research has been conducted on automatic program transformations that remove these intermediate structures, both for lazy and strict languages [1–4].

A common pattern that appears both in input programs and during supercompilation is a let-expression where the body is a case-expression: $\mathbf{let}\, x = e\, \mathbf{in\ case}\, e'\, \mathbf{of}\, \{p_i \rightarrow e_i\}$. A supercompiler for a strict language is only allowed to substitute e for x if we know that x is strict in the case-expression, and for pragmatic and proof technical reasons x must also be linear in the case-expression. As expected, it is quite easy to define functions that do not fulfill both of these requirements, or functions that are complex enough to fool the analyses used by the supercompiler. A standard example of such a function is *zip*.

If the supercompiler instead propagates let-expressions into the branches of case-expressions it simplifies the job for the analyses since they no longer need to account for different behaviours in different branches. Not only does this modification increase the precision of the analyses, but it also allows our improved supercompiler to remove more constructions that cause memory allocations. The propagation of let-expressions is orthogonal to the amount of information propagated, so it works for both positive [2] and perfect supercompilation [5]. We illustrate the increased strength through the following example:

$zip\ (map\ f_1\ xs)\ (map\ f_2\ ys)$

Its generalization to tree-like structures is also of interest:

$zipT\ (mapT\ f_3\ t_1)\ (mapT\ f_4\ t_2)$

These examples allow us to position supercompilation for a strict language relative to other well-known program transformations that perform program specialization and remove intermediate structures:

Shortcut deforestation [6] removes one of the intermediate lists in the first example, but does not remove arbitrary algebraic data types.

Stream fusion [7] removes both the intermediate lists in the first example, but does not remove arbitrary algebraic data types without manual extensions.

Positive supercompilation [2] for a strict language removes the first intermediate structure in both examples, and for a lazy language it removes both lists and both trees.

This paper presents one more step towards allowing the programmer to write clear and concise code in strict languages while getting good performance. The contributions of our work are:

- We provide a stronger algorithm for positive supercompilation in a strict and pure functional language (Section 4).
- We extend the supercompiler with a termination test that enables some unused let-bindings to be removed even though they are not fully evaluated. This feature is particularly beneficial in conjunction with the first contribution, since pushing bindings into case branches tend to result in many seemingly redundant let-expressions (Section 5).
- We prove the soundness of the algorithm in Section 6.

We start out with a step by step example where our improved supercompiler removes both intermediate lists for *zip* in Section 2 to give the reader an intuitive feel for how the algorithm behaves. Our language of study is defined in Section 3 followed by the technical contributions. We end with a discussion of the performance of the algorithm in Section 7.

2 Examples

This section gives a walk-through of the transformation of *zip* for readers who are already familiar with positive supercompilation for call-by-value [4]. For readers who are not at all familiar with these techniques there are more examples of step by step transformations in the work of Wadler [1] and Sørensen, Glück and Jones [2].

Our first example is transformation of the standard function *zip*, which takes two lists as parameters: $zip\ (map\ f\ xs')\ (map\ g\ ys')$. The standard definitions of *map* and *zip* are:

$$
\begin{array}{lll}
map & = \lambda f\ xs.\ \textbf{case}\ xs\ \textbf{of} & \\
 & \quad [] \ \rightarrow\ ys & \\
 & \quad (x : xs) \ \rightarrow\ f\ x : map\ f\ xs & \\
zip & = \lambda xs\ ys.\ \textbf{case}\ xs\ \textbf{of} & \\
 & \quad [] \ \rightarrow\ [] & \\
 & \quad (x' : xs') \ \rightarrow\ \textbf{case}\ ys\ \textbf{of} & \\
 & \qquad\qquad [] \ \rightarrow\ [] & \\
 & \qquad\qquad (y' : ys') \ \rightarrow\ (x', y') : zip\ xs'\ ys' &
\end{array}
$$

We start our transformation by allocating a new fresh function name (h_0) to this expression, inlining the body of *zip*, substituting $map\ f\ xs'$ into the body of *zip*, and putting $map\ g\ ys'$ into a let-expression to preserve termination properties of the program:

$$
\begin{array}{l}
\textbf{let}\ ys\ =\ map\ g\ ys' \\
\textbf{in case}\ map\ f\ xs'\ \textbf{of} \\
\quad [] \ \rightarrow\ [] \\
\quad (x' : xs') \ \rightarrow\ \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad [] \ \rightarrow\ [] \\
\qquad\qquad (y' : ys') \ \rightarrow\ (x', y') : zip\ xs'\ ys'
\end{array}
$$

The key difference between this algorithm and our previous work is that it transforms the case expression without touching the let-expression. After inlining the body of *map* in the head of the case-expression and substituting the arguments into the body the result becomes:

$$
\begin{array}{l}
\textbf{let}\ ys\ =\ map\ g\ ys' \\
\textbf{in case}\ (\ \textbf{case}\ xs'\ \textbf{of} \\
\qquad [] \ \rightarrow\ [] \\
\qquad (z : zs) \ \rightarrow\ f\ z : map\ f\ zs)\ \textbf{of} \\
\quad [] \ \rightarrow\ [] \\
\quad (x' : xs') \ \rightarrow\ \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad [] \ \rightarrow\ [] \\
\qquad\qquad (y' : ys') \ \rightarrow\ (x', y') : zip\ xs'\ ys'
\end{array}
$$

Notice how the let-expression is still untouched by the transformation – this is essential for the power of the transformation. We duplicate the let-expression and the outer case in each of the inner case's branches, using the expression in the branches as the head of the outer case-expression:

```
case xs' of
[] → let ys = map g ys'
     in case [] of
          [] → []
          (x' : xs') → case ys of
                         [] → []
                         (y' : ys') → (x', y') : zip xs' ys'
(z : zs) → let ys = map g ys'
           in case f z : map f zs of of
                [] → []
                (x' : xs') → case ys of
                               [] → []
                               (y' : ys') → (x', y') : zip xs' ys'
```

The case-expression in the first branch of the outermost case reduces to the empty list, but the let-expression must remain or we might introduce accidental termination in the program. The second branch is more interesting: we have a known constructor in the head of the case-expression so we can perform the reduction:

```
case xs' of
[] → let ys = map g ys' in []
(z : zs) → let ys = map g ys'
           in let x' = f z, xs' = map f zs
           in case ys of
                [] → []
                (y' : ys') → (x', y') : zip xs' ys'
```

The first branch can either be left as is, or one can transform the let-expression to get a new function that is isomorphic to *map* and a call to it. This is an orthogonal problem to removing multiple intermediate structures however, and we will not treat it further in this example. In Section 5 we show how to automatically remove superfluous let-expressions such as this through termination analysis. The reduction of the case-expression in the second branch reveals that the second branch is strict in *ys*, so *ys* will be evaluated, and the termination behavior will be the same even after performing the substitution. After performing the substitution we have:

```
case xs' of
[] → let ys = map g ys' in []
(z : zs) → let x' = f z, xs' = map f zs
           in case map g ys' of
                [] → []
                (y' : ys') → (x', y') : zip xs' ys'
```

We repeat inlining the body of *map* in the head of the inner case-expression and substituting the arguments into the body which gives:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
[]\ \rightarrow\ \textbf{let}\ ys\ =\ map\ g\ ys'\ \textbf{in}\ [] \\
(z : zs)\ \rightarrow\ \textbf{let}\ x'\ =\ f\ z,\ xs'\ =\ map\ f\ zs \\
\qquad\qquad \textbf{in case}\ (\ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \rightarrow [] \\
\qquad\qquad\qquad\qquad (z' : zs')\ \rightarrow\ g\ z' : map\ g\ zs')\ \textbf{of} \\
\qquad\qquad\qquad [] \rightarrow [] \\
\qquad\qquad\qquad (y' : ys')\ \rightarrow\ (x', y') : zip\ xs'\ ys'
\end{array}
$$

Once again we move the let-expression and the middle case into the branches of the innermost case:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
[]\ \rightarrow\ \textbf{let}\ ys\ =\ map\ g\ ys'\ \textbf{in}\ [] \\
(z : zs)\ \rightarrow\ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad [] \rightarrow\ \textbf{let}\ x'\ =\ f\ z,\ xs'\ =\ map\ f\ zs \\
\qquad\qquad\qquad\ \textbf{in case}\ []\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \rightarrow [] \\
\qquad\qquad\qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys' \\
\qquad\qquad (z' : zs')\ \rightarrow\ \textbf{let}\ x'\ =\ f\ z,\ xs'\ =\ map\ f\ zs \\
\qquad\qquad\qquad\qquad \textbf{in case}\ g\ z' : map\ g\ zs'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [] \rightarrow [] \\
\qquad\qquad\qquad\qquad\qquad (y' : ys')\ \rightarrow\ (x', y') : zip\ xs'\ ys'
\end{array}
$$

The first branch reduces to the empty list but we have to preserve the let-expression for termination purposes. Transforming the first branch is not going to reveal anything interesting, so we leave that branch as is, but of course the algorithm transforms that branch as well. The second branch is more interesting since it has a known constructor in the head of a case-expression, so we perform the reduction:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
[]\ \rightarrow\ \textbf{let}\ ys\ =\ map\ g\ ys'\ \textbf{in}\ [] \\
(z : zs)\ \rightarrow\ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad [] \rightarrow\ \textbf{let}\ x'\ =\ f\ z,\ xs'\ =\ map\ f\ zs\ \textbf{in}\ [] \\
\qquad\qquad (z' : zs')\ \rightarrow\ \textbf{let}\ x'\ =\ f\ z,\ xs'\ =\ map\ f\ zs \\
\qquad\qquad\qquad\qquad \textbf{in}\ (x', g\ z') : zip\ xs'\ (map\ g\ zs')
\end{array}
$$

After the reduction it is clear that both *x'* and *xs'* are really strict, so it is safe to substitute them:

$$
\begin{array}{l}
\textbf{case}\ xs'\ \textbf{of} \\
[]\ \rightarrow\ \textbf{let}\ ys\ =\ map\ g\ ys'\ \textbf{in}\ [] \\
(z : zs)\ \rightarrow\ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad [] \rightarrow\ \textbf{let}\ x'\ =\ f\ z,\ xs'\ =\ map\ f\ zs\ \textbf{in}\ [] \\
\qquad\qquad (z' : zs')\ \rightarrow\ (f\ z, g\ z') : zip\ (map\ f\ zs)\ (map\ g\ zs')
\end{array}
$$

We notice a familiar expression in $zip\ (map\ f\ zs)\ (map\ g\ zs')$, which is a renaming of what we started with, and fold here. This gives a new function h_0 and a call to that function as a final result:

```
letrec h0 = λ f xs' g ys'.
  case xs' of
   [] → let ys = map g ys' in []
   (z : zs) → case ys' of
                [] → let x' = f z, xs' = map f zs in []
                (z' : zs') → (f z, g z') : h0 f zs g zs'
in h0 f xs' g ys'
```

The new function h_0 does not pass any intermediate lists for the common case when both *xs* and *ys* are non-empty. If one of them is empty, it is necessary to run *map* on the remaining part of the other list.

In the introduction we claimed that we can fuse both intermediate lists and both intermediate trees when zipping a list or a tree. The second example requires some new definitions of *map* and *zip* over a simple tree datatype:

```
data Tree a = Node (Tree a) a (Tree a) | Empty

mapT = λf xs. case xs of
                Empty → Empty
                Node l a r → Node (mapT f l) (f a) (mapT f r)
zipT = λxs ys.
       case xs of
         Empty → Empty
         Node l a r →
             case ys of
               Empty → Empty
               Node l' a' r' → Node (zipT l l') (a, a') (zipT r r')
```

We transform the expression $zipT\ (mapT\ f\ xs)\ (mapT\ g\ ys)$, which applies f to the first tree, g to the second tree and create a final tree whose nodes consists of pairs of the data from the two intermediate trees. We start our transformation by allocating a new fresh function name (h_1) and repeat many of the transformation steps that we just saw for the list case. The end result is:

```
letrec h1 = λ f xs g ys.
  case xs of
   Empty → let ys' = mapT g ys in Empty
   Node l a r →
     case ys of
      Empty → let l1 = mapT f l, a1 = f a
                  r1 = mapT f r in Empty
      Node l' a' r' → Node (h1 f l g l') (f a, g a') (h1 f r g r')
in h1 f xs g ys
```

The same result as in the list case: the new function h_1 does not pass any intermediate trees for the common case: when both *xs* and *ys* are non-empty. If one of them is empty, it is necessary to run $mapT$ on the remaining part of the other tree. This example also highlights the need to discard unused let-bindings.

The third example of how the new algorithm improves the strength of supercompilation for call-by-value languages is non-linear occurrences of variables, such as in

$$\textbf{let}\ x\ =\ e\ \textbf{in}\ \mathit{fst}\ (x,\ x)$$

Our previous algorithm would separately transform e and *fst (x, x)* which would result in $\textbf{let}\ x\ =\ e'\ \textbf{in}\ x$, where it is obvious that x is linear. Our improved algorithm instead inlines *fst* without touching e:

$$\textbf{let}\ x\ =\ e\ \textbf{in case}\ (x,\ x)\ \textbf{of}\ \{\ (x,\ y)\ \rightarrow\ x\ \}$$

The algorithm continues to transform the case-expression giving a let-expression that is linear in x: $\textbf{let}\ x\ =\ e\ \textbf{in}\ x$. This expression can be transformed to e and the supercompiler can continue to transform e, having eliminated the entire let-expression in the initial program.

3 Language

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

Expressions

$$e,\ f\ ::=\ x\ \mid\ g\ \mid\ k\,\overline{e}\ \mid\ f\,e\ \mid\ \lambda x.e\ \mid\ \textbf{case}\,e\ \textbf{of}\ \{k_i\,\overline{x}_i \rightarrow e_i\}\ \mid\ \textbf{let}\,x = e\ \textbf{in}\,f \\ \mid\ \textbf{letrec}\,g = v\,\textbf{in}\,e$$

Values

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

Fig. 1. The language

Let X be an enumerable set of variables ranged over by x and K a set of constructor symbols ranged over by k. Let g range over an enumerable set of defined names and let $\mathcal{G}$ be a given set of recursive definitions of the form (g, v).

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 $\overline{x}$. We denote the set of ordered free variables of an expression e by $\mathit{fv}(e)$, and the function names by $\mathit{fn}(e)$.

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 2, 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

Reduction contexts

$$\mathcal{E} ::= [\,] \mid \mathcal{E}\, e \mid (\lambda x.e)\, \mathcal{E} \mid k\, \overline{\mathcal{E}} \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ \{p_i \to e_i\} \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ e$$

Evaluation relation

$$\begin{array}{lll}
\mathcal{E}\langle g \rangle & \mapsto \mathcal{E}\langle v \rangle & \text{(Global)} \\
 & \quad \text{if } (g, v) \in \mathcal{G} & \\
\mathcal{E}\langle (\lambda x.e)\, v \rangle & \mapsto \mathcal{E}\langle [v/x]e \rangle & \text{(App)} \\
\mathcal{E}\langle \mathbf{let}\ x = v\ \mathbf{in}\ e \rangle & \mapsto \mathcal{E}\langle [v/x]e \rangle & \text{(Let)} \\
\mathcal{E}\langle \mathbf{letrec}\ g = v\ \mathbf{in}\ f \rangle & \mapsto \mathcal{E}\langle [\mathbf{letrec}\ g = v\ \mathbf{in}\ v/g]f \rangle & \text{(Letrec)} \\
\mathcal{E}\langle \mathbf{case}\ k_j\ \overline{v}\ \mathbf{of}\ \{k_i\ \overline{x}_i \to e_i\} \rangle & \mapsto \mathcal{E}\langle [\overline{v}/\overline{x}_j]e_j \rangle & \text{(KCase)}
\end{array}$$

Fig. 2. Reduction semantics

replacing the hole in $\mathcal{E}$ with e. $\overline{\mathcal{E}}$ denotes a list of terms with just a single hole, evaluated from left to right.

If a free variable appears no more than once in a term, that term is said to be *linear* with respect to that variable. Like Wadler [1], we extend the definition slightly for linear case expressions: no variable may appear in both the scrutinee and a branch, although a variable may appear in more than one branch.

4 Positive Supercompilation

This section presents an algorithm for positive supercompilation for a higher-order call-by-value language, which removes more intermediate structures than previous work [4].

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 3. Three additional parameters appear as subscripts to the rewrite rules: a memoization list ρ, a driving context $\mathcal{R}$, and an ordered set $\mathcal{B}$ of expressions bound to variables ($x_1 = e_1,\ x_2 = e_2, \ldots$). We use the short-hand notation $\mathbf{let}\ \mathcal{B}\ \mathbf{in}\ e$ for $\mathbf{let}\ x_1 = e_1\ \mathbf{in let}\ x_2 = e_2\ \mathbf{in} \ldots \mathbf{in}\ e$. The memoization list holds information about expressions already traversed and is explained more in detail in Section 4.1. An important detail is that our driving algorithm immediately performs the program extraction instead of producing a process tree. The driving context $\mathcal{R}$ is an evaluation context for a call-by-name language:

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

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 decidable in general, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 4, where the strict variables of an expression e are defined as all free variables of e except those that only

$$
\begin{array}{lll}
\mathcal{D}[\![x]\!]_{[],\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\ \mathcal{D}[\![\mathcal{B}]\!]_{[],\emptyset,\mathcal{G},\rho}\ \textbf{in}\ x & \text{(R1)} \\
\mathcal{D}[\![g]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}_{app}(g)_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & \text{(R2)} \\
\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\ \mathcal{D}[\![\mathcal{B}]\!]_{[],\emptyset,\mathcal{G},\rho}\ \textbf{in}\ k\,\mathcal{D}[\![\overline{e}]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R3)} \\
\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\ \mathcal{D}[\![\mathcal{B}]\!]_{[],\emptyset,\mathcal{G},\rho}\ \textbf{in}\ \mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\emptyset,\mathcal{G},\rho}\rangle & \text{(R4)} \\
\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{B},\mathcal{G},\rho} & = (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{B},\mathcal{G},\rho}) & \text{(R5)} \\
\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\ \overline{x} = \overline{e}\ \textbf{in}\ f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & \text{(R6)} \\
\mathcal{D}[\![e\,e']\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,e'\rangle,\mathcal{B},\mathcal{G},\rho} & \text{(R7)} \\
\mathcal{D}[\![\textbf{let}\ x = v\ \textbf{in}\ f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\ \mathcal{B}\ \textbf{in}\ \mathcal{R}\langle[v/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R8)} \\
\mathcal{D}[\![\textbf{let}\ x = y\ \textbf{in}\ f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\ \mathcal{B}\ \textbf{in}\ \mathcal{R}\langle[y/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R9)} \\
\mathcal{D}[\![\textbf{let}\ x = e\ \textbf{in}\ f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\ \mathcal{B}\ \textbf{in}\ \mathcal{R}\langle[e/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho},\ \text{if } x \in \mathit{strict}(f) & \text{(R10)} \\
 & \qquad\qquad \text{and } x \in \mathit{linear}(f) & \\
 & \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{B}\oplus x=e,\mathcal{G},\rho},\ \text{otherwise} & \\
\mathcal{D}[\![\textbf{letrec}\ g = v\ \textbf{in}\ e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{letrec}\ g = v\ \textbf{in}\ e',\ \text{if } g \in \mathit{fn}(e') & \text{(R11)} \\
 & e',\ \text{otherwise} & \\
 & \text{where } e' = \mathcal{D}[\![\textbf{let}\ \mathcal{B}\ \textbf{in}\ \mathcal{R}\langle e\rangle]\!]_{[],\emptyset,\mathcal{G}',\rho} & \\
 & \qquad \mathcal{G}' = \mathcal{G} \cup (g, v) & \\
\mathcal{D}[\![\textbf{case}\ x\ \textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\ \mathcal{D}[\![\mathcal{B}|x]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R12)} \\
 & \quad \textbf{in case}\ x\ \textbf{of}\ \{ & \\
 & \qquad k_i\,\overline{x}_i \to \mathcal{D}[\![[k_i\,\overline{x}_i/x]\textbf{let}\ \mathcal{B}\backslash x\ \textbf{in}\ \mathcal{R}\langle e_i\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \\
 & \quad \} & \\
\mathcal{D}[\![\textbf{case}\ k_j\,\overline{e}\ \textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\ \mathcal{B}\ \textbf{in}\ \mathcal{R}\langle\textbf{let}\ \overline{x}_j = \overline{e}\ \textbf{in}\ e_j\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R13)} \\
\mathcal{D}[\![\textbf{case}\ x\,\overline{e}\ \textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\ \mathcal{D}[\![\mathcal{B}|(\mathit{fv}(\overline{e}) \cup \{x\})]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R14)} \\
 & \quad \textbf{in case}\ x\ \mathcal{D}[\![\overline{e}]\!]_{[],\emptyset,\mathcal{G},\rho}\ \textbf{of}\ \{ & \\
 & \qquad k_i\,\overline{x}_i \to \mathcal{D}[\![\textbf{let}\ \mathcal{B}\backslash(\mathit{fv}(\overline{e}) \cup \{x\})\ \textbf{in}\ \mathcal{R}\langle e_i\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \\
 & \quad \} & \\
\mathcal{D}[\![\textbf{case}\ e\ \textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![e]\!]_{\mathcal{R}\langle\textbf{case}\ []\ \textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}\rangle,\mathcal{B},\mathcal{G},\rho} & \text{(R15)}
\end{array}
$$

Fig. 3. Driving algorithm

appear under a lambda or not inside all branches of a case. Our experience is that this approximation is sufficient in practice.

The rules of the driving algorithm are ordered; i.e., all rules must be tried in the order they appear. Rule R7 and rule R15 are the default fallback cases which extend the given driving context $\mathcal{R}$ and zoom in on the next expression to drive. Notice how rule R8 recursively applies the driving algorithm to the entire new term $\textbf{let}\ \mathcal{B}\ \textbf{in}\ \mathcal{R}\langle[v/x]f\rangle$, forcing a re-traversal of the new term with the hope of further reductions.

Some expressions should be handled differently depending on their context. If a constructor application appears in an empty context, there is not much we can do except to drive the argument expressions (rule R3). On the other hand, if the application occurs at the head of a case expression, we may choose a branch based on the constructor and leave the arguments unevaluated in the hope of finding fold opportunities further down the syntax tree (rule R13).

Rule R12 and rule R14 uses some new notation: $\mathcal{B}|x$ is the smallest prefix of $\mathcal{B}$ that is necessary to define x and $\mathcal{B}\backslash x$ is the largest suffix not necessary to

$$
\begin{array}{ll}
\mathit{strict}(x) & = \{x\} \\
\mathit{strict}(g) & = \emptyset \\
\mathit{strict}(k\,\overline{e}) & = \mathit{strict}(\overline{e}) \\
\mathit{strict}(\lambda x.e) & = \emptyset \\
\mathit{strict}(f\,e) & = \mathit{strict}(f) \cup \mathit{strict}(e) \\
\mathit{strict}(\mathbf{let}\,x = e\,\mathbf{in}\,f) & = \mathit{strict}(e) \cup (\mathit{strict}(f) \backslash \{x\}) \\
\mathit{strict}(\mathbf{letrec}\,g = v\,\mathbf{in}\,f) & = \mathit{strict}(f) \\
\mathit{strict}(\mathbf{case}\,e\,\mathbf{of}\,\{k_i\,\overline{x}_i \to e_i\}) & = \mathit{strict}(e) \cup (\bigcap(\mathit{strict}(e_i) \backslash \overline{x}_i))
\end{array}
$$

Fig. 4. The strict variables of an expression

define x. Rule R10 uses $\oplus$ which we define as:

$$
\begin{aligned}
(\mathcal{B}, y = e) \oplus x = e' &\stackrel{\text{def}}{=} (\mathcal{B} \oplus x = e'), y = e \text{ if y} \notin \mathit{fv}(e') \\
&\stackrel{\text{def}}{=} (\mathcal{B}, y = e, x = e') \text{ otherwise}
\end{aligned}
$$

The key idea in this improved supercompilation algorithm is to float let-expressions into the branches of case-expressions. We accomplish this by adding the bound expressions from let-expressions to our binding set $\mathcal{B}$ in rule R10. We make sure that we do not change the order between definition and usage of variables in rule R8 by extracting the necessary bindings outside of the case-expression, and the remaining independent bindings are brought into all the branches along with the surrounding context $\mathcal{R}$.

The algorithm is allowed to move let-expressions into case-branches since that transformation only changes the evaluation order, and non-termination is the only effect present in our language.

4.1 Application Rule

Our extension does not require any major changes to the folding mechanism that supercompilers use to ensure termination. Since our goal is not to study termination properties of supercompilers we present a simplified version of the folding mechanism which does not guarantee termination, but guarantees that if the algorithm terminates the output is correct. The standard techniques [8, 9, 4] for ensuring termination can be used with our extension.

In the driving algorithm rule R2 refer to $\mathcal{D}_{app}(\,)$, defined in Figure 5. $\mathcal{D}_{app}(\,)$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation. Figure 5 contains some new notation: we use σ for a variable to variable substitution and $=$ for syntactical equivalence of expressions.

The driving algorithm keeps a record of previously encountered 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 an expression from the memoization list is encountered, a call to h_n is inserted.

$$\mathcal{D}_{app}(g)_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} = h\,\overline{x} \qquad \text{if } \exists (h,t) \in \rho \,.\, \sigma t = \mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle g\rangle \quad (1)$$
where $\overline{x} = \sigma(fv(t))$

$$\mathcal{D}_{app}(g)_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} = \mathbf{letrec}\, h = \lambda \overline{x}.e'\,\mathbf{in}\, h\,\overline{x} \qquad \text{if } h \in fn(e') \quad (2a)$$
$$e' \qquad \text{otherwise} \quad (2b)$$
where $(g, v) \in \mathcal{G}$,
$e' = \mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{[],\mathcal{B},\mathcal{G},\rho'}$,
$\overline{x} = fv(\mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle g\rangle)$,
$\rho' = \rho \cup (h, \mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle g\rangle)$ and
h fresh

Fig. 5. Driving of applications

5 Removing Unnecessary Traversals

The first example in Section 2 showed that there might be let-expressions in case-branches where the computed results are never used in the branch. This gives worse runtime performance than necessary since more intermediate results have to be computed, and also increases the compilation time since there are more expressions to transform. The only reason to have these let-expressions is to preserve the termination properties of the input program.

We could remove these superfluous let-expressions if we knew that they were terminating, something that would save both transformation time and execution time. It is clear that termination is undecidable in general, but our experience is that the functions that appear in practice are often recursive over the input structure. Functions with this property are quite well suited for termination analysis, for example the size-change principle [10, 11].

Given a function *terminates*(e) that returns true if the expression e terminates, we can augment the let-rule (R10) to incorporate the termination information and discard such expressions, shown in Figure 6. This allows the supercompiler to discard unused expressions, i.e dead code, which saves both transformation time and runtime.

$$\mathcal{D}[\![\mathbf{let}\, x = e\,\mathbf{in}\, f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} = \begin{cases} \mathcal{D}[\![\mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{if } \textit{terminates}(\text{e}) \text{ and } x \notin fv(f) \\ \mathcal{D}[\![\mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{if } x \in \textit{strict}(f) \text{ and } x \in \textit{linear}(f) \\ \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{B}\oplus x=e,\mathcal{G},\rho} & \text{otherwise} \end{cases}$$

Fig. 6. Extended Let-rule (R10)

Since we leave the choice of termination analysis open, it is hard to discuss scalability in general. The size-change principle has been used with good results in partial evaluation of large logic programs [12] and there are also polynomial time algorithms for approximating termination [13].

6 Correctness

We need a class of expressions w that is essentially the union of the expressions on weak head normal form (whnf) and expressions where the redex contains a free variable, a:

$$a ::= x \mid a\,\overline{e}$$
$$w ::= \lambda x.e \mid k\,\overline{e} \mid a$$

Lemma 1 (Totality). *Let $\mathcal{R}\langle e\rangle$ be an expression such that*

- *$\mathcal{R}\langle e\rangle$ is well-typed*
- *if $\mathcal{R} \neq [\,]$ then $e \neq w$*

then $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho}$ is matched by a unique rule in Figure 3.

Proof. Follows the structure of the proof by Jonsson and Nordlander [14].

To prove that the algorithm does not alter the semantics we use the improvement theory [15]. We define the standard notions of operational approximation and equivalence and introduce a general context C which has zero or more holes in the place of some subexpressions.

Definition 1 (Operational Approximation and Equivalence).

- *e operationally approximates e', $e \mathrel{\underset{\sim}{\sqsubset}} e'$, if for all contexts C such that C[e] and 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 $e \mathrel{\underset{\sim}{\sqsubset}} e'$ and $e' \mathrel{\underset{\sim}{\sqsubset}} e$*

We use Sands's definitions for improvement and strong improvement:

Definition 2 (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 calls to named functions, then computation of C[e'] also terminates, and uses no more than n calls to named functions.*
- *e is strongly improved by e', $e \trianglerighteq_s e'$, iff $e \trianglerighteq e'$ and $e \cong e'$.*

which allows us to state the final theorem:

Theorem 1 (Total Correctness). *Let $\mathcal{R}\langle e\rangle$ be an expression, and ρ an environment such that*

- *the range of ρ contains only closed expressions, and*
- *$fv(\mathcal{R}\langle e\rangle) \cap dom(\rho) = \emptyset$, and*
- *if $\mathcal{R} \neq [\,]$ then $e \neq w$*
- *the supercompiler $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho}$ terminates*

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

Proof. Similar to the total correctness proof by Jonsson and Nordlander [14].

Recall the simplified algorithm we have presented preserves the semantics only if it terminates; however, termination of the supercompiler can be recovered using a similar $\mathcal{D}_{app}(\,)$ as we did in our previous work [4].

7 Performance and Limitations

There are two aspects of performance that are interesting to the end user: how long the optimization takes; and how much faster the optimized program is.

The work on supercompiling Haskell by Mitchell and Runciman [9] shows that some problems remain for supercompiling large Haskell programs. These problems are mainly related to speed, both of the compiler and of the transformed program. When they profiled their supercompiler they found that the majority of the time was spent in the homeomorphic embedding test, the test which is used to ensure termination.

Our preliminary measurements show the same thing: a large proportion of the time spent on supercompiling a program is spent testing for non-termination of the supercompiler. This paper presents a stronger supercompiler at the cost of larger expressions to test for the homeomorphic embedding. We estimate that our current work ends up somewhere between Supero and our previous work with respect to transformation time since we are testing smaller expressions than Supero, at the expense of runtime performance.

The complexity of the homeomorphic embedding has been investigated separately by Narendran and Stillman [16] and they give an algorithm that takes two terms e and f and decides if there is a risk of non-termination in time $O(size(e) \times size(f))$.

For the second dimension: it is well known that programs with many intermediate lists have worse performance than their corresponding listless versions [6]. We have shown that the output from our supercompiler does not contain intermediate structures by manual transformations in Section 2. It is reasonable to conclude that these programs would perform better in a microbenchmark. We leave the question of performance of large real world programs open.

A limitation of our work is that there are still examples that our algorithm does not give the desired output for. Given $\mathbf{let}\ x\ =(\lambda y.y)\ 1\ \mathbf{in}\ (x,\ x)$ a human can see that the result after transformation should be (1, 1), but our supercompiler will produce $\mathbf{let}\ x\ =\ 1\ \mathbf{in}\ (x,\ x)$. Mitchell [17][Sec 4.2.2] has a strategy to handle this, but we have not been able to incorporate his solution without severely increasing the amount of testing for non-termination done with the homeomorphic embedding. The reason we can not transform $(\lambda y.y)\ 1$ in isolation and then substitute the result is that the result might contain freshly generated function names, which might cause the supercompiler to loop.

8 Related Work

8.1 Deforestation

Deforestation is a slightly weaker transformation than supercompilation [18]. Deforestation algorithms for call-by-name languages can remove all intermediate structures from the examples we outlined in Section 1.

Deforestation was pioneered by Wadler [1] for a first order language more than fifteen years ago. The initial deforestation had support for higher order macros which are incapable of fully emulating higher order functions.

Marlow and Wadler [19] 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 [20] dissertation, where he also related deforestation to the cut-elimination principle of logic. Chin [21] has also generalised Wadler's deforestation to higher-order functional programs by using syntactic properties to decide which terms can be fused.

Both Hamilton [22] and Marlow [20] have proven that their deforestation algorithms terminate. More recent work by Hamilton [23] 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 [24] have improved the producer and consumer analyses in Chin's [21] algorithm by basing them on semantics rather than syntax. They show that their algorithm can remove much of the overhead introduced from generic programming [25].

8.2 Supercompilation

Except for our previous work [4], the work on supercompilation has been for call-by-name semantics. All call-by-name supercompilers succeed on the examples we outlined in Section 1, and are very close algorithmically to our current work.

Supercompilation [26–29] removes intermediate structures and achieves partial evaluation as well as some other optimisations. Scp4 [30] is the most well-known implementation from this line of work.

The *positive supercompiler* [2] 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. We have previously investigated the theoretical foundations for positive supercompilation for strict languages [4]. Narrowing-driven partial evaluation [31, 32] is the functional logic programming equivalent of positive supercompilation but formulated as a term rewriting system. They also deal with non-determinism 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* [5, 33, 34]. Negative information is the opposite of positive information – inequalities. These inequalities can for example be used to prune branches that we can be certain are not taken in case-expressions.

More recently, Mitchell and Runciman [9] have worked on supercompiling Haskell. Their algorithm is closely related to our supercompiler, but their work is limited to call-by-name. They report runtime reductions of up to 55% when their supercompiler is used in conjunction with GHC.

8.3 Short Cut Fusion

Short cut deforestation [35, 6] 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 to allow the transformation to recognize the producer and enforce the type requirement. Lists using *build/foldr* can easily be removed with the *foldr/build* rule: $foldr\ f\ c\ (build\ g)\ =\ g\ f\ c$.

This forces 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.

Takano and Meijer [36] generalized short cut deforestation to work for any algebraic datatype through the acid rain theorem. Ghani and Johann [37] have also 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.

Launchbury and Sheard [38] worked on automatically transforming programs into suitable form for shortcut deforestation. Onoue et al. [39] showed an implementation of the acid rain theorem for Gofer where they could automatically transform recursive functions into a form suitable for shortcut fusion.

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 [40] 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.

Takano and Meijer [36] noted that the foldr/build rule for short cut deforestation had a dual. This is the *destroy/unfoldr* rule used by Svenningsson [41] which has some interesting properties. The method can remove all argument lists from a function which consumes more than one list, addressing 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 that most techniques can not handle.

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*.

In more recent work Coutts et al. [7] have extended these techniques to work on functions that handle nested lists, list comprehensions and filter-like functions.

9 Conclusions

We have presented an improved supercompilation algorithm for a higher-order call-by-value language. Our extension is orthogonal to the information propagation by the algorithm. Through examples we have shown that the algorithm can

remove multiple intermediate structures, which previous algorithms could not, such as the *zip* examples in Section 2.

9.1 Future Work

We are currently working on improving the scalability of supercompilation for real programs. MLTon [42] has successfully performed whole program compilation of programs up to 100 000 lines, which suggests that any bottlenecks should occur in the supercompiler, not the other parts of the compiler.

An anonymous referee suggested performing on-demand termination analysis on already simplified terms. We are looking into this possibility. Another anonymous referee suggested a characterization, such as the treeless form by Wadler [1], of input terms that would guarantee termination of the supercompiler as specified in this paper.

Acknowledgements The authors would like to thank Germán Vidal for sharing his practical experience with the size-change principle and Duncan Coutts for answering our questions about stream fusion. We would also like to thank Viktor Leijon and the anonymous referees for providing useful comments that helped improve the presentation and contents.

References

1. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theoretical Computer Science **73**(2) (June 1990) 231–248
2. Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming **6**(6) (1996) 811–838
3. Ohori, A., Sasano, I.: Lightweight fusion by fixed point promotion. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2007) 143–154
4. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher-order call-by-value language. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. (2009)
5. Glück, R., Klimov, A.: Occam's razor in metacomputation: the notion of a perfect process tree. Lecture Notes in Computer Science **724** (1993) 112–123
6. Gill, A.J.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis, Univ. of Glasgow (January 1996)
7. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: from lists to streams to nothing at all. In: ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2007) 315–326
8. Sørensen, M., Glück, R.: An algorithm of generalization in positive supercompilation. In Lloyd, J., ed.: International Logic Programming Symposium, Cambridge, MA: MIT Press (1995) 465–479
9. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In et al., O.C., ed.: Selected Papers from the Proceedings of IFL 2007. Volume 5083 of Lecture Notes in Computer Science., Springer-Verlag (2008) 147–164

10. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL'01. (2001) 81–92
11. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In Hinze, R., Ramsey, N., eds.: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007, ACM (2007) 71–84
12. Leuschel, M., Vidal, G.: Fast Offline Partial Evaluation of Large Logic Programs. In: Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'08), Springer LNCS 5438 (2009) 119–134
13. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. ACM Trans. Program. Lang. Syst **29**(1) (2007)
14. Jonsson, P.A., Nordlander, J.: 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)
15. Sands, D.: 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)
16. Narendran, P., Stillman, J.: On the Complexity of Homeomorphic Embeddings. Technical Report 87-8, Computer Science Department, State Univeristy of New York at Albany (March 1987)
17. Mitchell, N.: Transformation and Analysis of Functional Programs. PhD thesis, University of York (June 2008)
18. Sørensen, M., Glück, R., Jones, N.: Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Sannella, D., ed.: Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788), Berlin: Springer-Verlag (1994) 485–500
19. Marlow, S., Wadler, P.: Deforestation for higher-order functions. In Launchbury, J., Sansom, P.M., eds.: Functional Programming. Workshops in Computing, Springer (1992) 154–165
20. Marlow, S.D.: Deforestation for Higher-Order Functional Programs. PhD thesis, Department of Computing Science, University of Glasgow (April 27 1995)
21. Chin, W.N.: Safe fusion of functional expressions II: Further improvements. J. Funct. Program **4**(4) (1994) 515–555
22. Hamilton, G.W.: Higher order deforestation. In: PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, London, UK, Springer-Verlag (1996) 213–227
23. Hamilton, G.W.: Higher order deforestation. Fundam. Informaticae **69**(1-2) (2006) 39–61
24. Alimarine, A., Smetsers, S.: Improved fusion for optimizing generics. In Hermenegildo, M.V., Cabeza, D., eds.: 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., Springer (2005) 203–218
25. Hinze, R.: Generic Programs and Proofs. Habilitationsschrift, Bonn University (2000)
26. Turchin, V.: A supercompiler system based on the language Refal. SIGPLAN Notices **14**(2) (February 1979) 46–54

27. Turchin, V.: Semantic definitions in Refal and automatic production of compilers. In Jones, N., ed.: Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94). Berlin: Springer-Verlag (1980) 441–474
28. Turchin, V.: Program transformation by supercompilation. In Ganzinger, H., Jones, N., eds.: Programs as Data Objects, Copenhagen, Denmark, 1985 (Lecture Notes in Computer Science, vol. 217). Berlin: Springer-Verlag (1986) 257–281
29. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems **8**(3) (July 1986) 292–325
30. Nemytykh, A.P.: The supercompiler SCP4: General structure. In Broy, M., Zamulin, A.V., eds.: Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers. Volume 2890 of LNCS., Springer (2003) 162–170
31. Alpuente, M., Falaschi, M., Vidal, G.: Partial Evaluation of Functional Logic Programs. ACM Transactions on Programming Languages and Systems **20**(4) (1998) 768–844
32. Albert, E., Vidal, G.: The narrowing-driven approach to functional logic program specialization. New Generation Comput **20**(1) (2001) 3–26
33. Secher, J.P.: Perfect supercompilation. Technical Report DIKU-TR-99/1, Department of Computer Science (DIKU), University of Copenhagen (February 1999)
34. Secher, J., Sørensen, M.: On perfect supercompilation. In Bjørner, D., Broy, M., Zamulin, A., eds.: Proceedings of Perspectives of System Informatics. Volume 1755 of Lecture Notes in Computer Science., Springer-Verlag (2000) 113–127
35. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 1993. (1993)
36. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: FPCA. (1995) 306–313
37. Ghani, N., Johann, P.: Short cut fusion of recursive programs with computational effects. In Achten, P., Koopman, P., Morazán, M.T., eds.: Draft Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP). Number ICIS–R08007 (2008)
38. Launchbury, J., Sheard, T.: Warm fusion: Deriving build-cata's from recursive definitions. In: FPCA. (1995) 314–323
39. Onoue, Y., Hu, Z., Iwasaki, H., Takeichi, M.: A calculational fusion system HYLO. In Bird, R.S., Meertens, L.G.L.T., eds.: Algorithmic Languages and Calculi, IFIP TC2 WG2.1 International Workshop on Algorithmic Languages and Calculi, 17-22 February 1997, Alsace, France. Volume 95 of IFIP Conference Proceedings., Chapman & Hall (1997) 76–106
40. Chitil, O.: Type-Inference Based Deforestation of Functional Programs. PhD thesis, RWTH Aachen (October 2000)
41. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: ICFP. (2002) 124–132
42. Weeks, S.: Whole-program compilation in MLton. In: ML '06: Proceedings of the 2006 workshop on ML, New York, NY, USA, ACM (2006) 1–1 http://mlton.org/pages/References/attachments/060916-mlton.pdf.