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

Peter A. Jonsson Johan Nordlander
Luleå University of Technology
{pj, nordlander}@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 runtime 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.

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 \( \text{append} (\text{append} \, x \, y) \, z \) and shows that his deforestation algorithm transforms the program so that it saves one traversal of the first list, thereby reducing the complexity from \( 2|x| + |y| \) to \( |x| + |y| \).

\[ \text{append} \, x \, y \, z = \text{case} \, x \, \text{of} \]
\[ \quad [] \rightarrow y \]
\[ \quad (x : z) \rightarrow x : \text{append} \, x \, y \, z \]

If we naively change Wadler’s algorithm to call-by-value semantics by eagerly attempting to transform arguments before at- tempting to transform the arguments to a function past inlining of its body — actually leads to the same result as Wadler obtains after transformation of arguments to a function past inlining of its body. The intermediate structure from the input program is still there after transformation, and the complexity remains at \( 1 + |x| + |y| \).

The end result from this transformation is:

\[ f = \text{append} \, (\text{append} \, x' \, y') \, z' \]

(Inline the body of \( \text{append} \, x' \, y' \) in the context \( \text{append} \, [] \, z' \) and push down the context into each branch of the case)

\[ f = \text{case} \, x' \, \text{of} \]
\[ \quad [] \rightarrow \text{append} \, y' \, z' \]
\[ \quad (x : z) \rightarrow \text{append} \, (x : \text{append} \, x \, y') \, z' \]

(Transform each branch but focus on the \((x:z)\) case)

\[ \text{append} \, (x : \text{append} \, x \, y') \, z' \]

(The expression contains an expression that has already been seen. The subexpression \( x : \text{append} \, x \, y' \) is extracted for separate transformation)

\[ x : \text{append} \, x \, y' \]
\[ x : \text{case} \, x \, \text{of} \]
\[ \quad [] \rightarrow y' \]
\[ \quad (x' : z') \rightarrow x' : \text{append} \, x \, y' \]

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

The end result from this transformation is:

\[ f = \text{case} \, x' \, \text{of} \]
\[ \quad [] \rightarrow h_1 \, y' \, z' \]
\[ \quad (x : z) \rightarrow h_1 \, (h_2 \, x \, x \, y') \, z' \]

\[ h_1 \, x \, y \, z = \text{case} \, x \, \text{of} \]
\[ \quad [] \rightarrow y \]
\[ \quad (x' : z') \rightarrow h_2 \, x \, y \]

\[ h_2 \, x \, x \, y \, z = x : \text{case} \, x \, \text{of} \]
\[ \quad [] \rightarrow y \]
\[ \quad (x' : z') \rightarrow h_2 \, x' \, y' \]

The intermediate structure from the input program is still there after the transformation, and the complexity remains at \( 2|x| + |y| + |z| \)!

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 \( \text{append} \, (\text{append} \, x \, y) \, z \). 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 \( \text{sum} \, (\text{map} \, \text{square} \, y) \). The functions used in the examples are defined as:

\[ \text{square} \, x = x \times x \]
\[ \text{map} \, f \, x = \text{case} \, x \, \text{of} \]
\[ \quad [] \rightarrow y \]
\[ \quad (x : z) \rightarrow f \, x : \text{map} \, f \, z \]
\[ \text{sum} \, x = \text{case} \, x \, \text{of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x : z) \rightarrow x + \text{sum} \, z \]

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

\[ \text{case} \, \text{map} \, \text{square} \, y \, \text{of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x' : z') \rightarrow x' + \text{sum} \, z' \]

After inlining \( \text{map} \) and substituting the arguments into the body the result becomes:

\[ \text{case} \, (\text{case} \, y) \, \text{of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x' : z') \rightarrow (\text{square} \, x') : \text{map} \, \text{square} \, z' \]

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:

\[ \text{case} \, y \, \text{of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x' : z') \rightarrow \text{square} \, x' + \text{sum} \, (\text{map} \, \text{square} \, z') \]

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 \( h_0 \, x' \). The result of our transformation is \( h_0 \, y \), with \( h_0 \) defined as:

\[ h_0 \, y = \text{case} \, y \, \text{of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x' : z') \rightarrow x' + h_0 \, z' \]

This new function only traverses its input once, and no intermediate structures are created. If the expression \( \text{sum} \, (\text{map} \, \text{square} \, x) \) 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) cannot 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:
\[
\text{mapsq } xs = \text{case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (x' \ast x') \text{ : mapsq } xs'
\end{cases}
\]

\[
f xs = \text{case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (2 \ast x') \text{ : g } xs'
\end{cases}
\]

\[
g xs = \text{case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (3 \ast x') \text{ : f } xs'
\end{cases}
\]

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

\[
\text{case } (\text{ case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (x' \ast x') \text{ : mapsq } xs'
\end{cases}) \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (x' \ast x') \text{ : mapsq } xs'
\end{cases}
\]

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:

\[
\text{case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (x' \ast x' \ast x' \ast x') \text{ : mapsq } (\text{mapsq } xs')
\end{cases}
\]

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 \(h_1 \text{ xs}\), with the new residual function \(h_1\) that only traverses its input once defined as:

\[
h_1 \text{ xs} = \text{case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (x' \ast x' \ast x' \ast x') \text{ : h}_1 \text{ xs'}
\end{cases}
\]

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

\[
\text{case } (\text{ case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow (2 \ast x') \text{ : g } xs'
\end{cases}) \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow x' + \text{sum } xs'
\end{cases}
\]

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

\[
\text{case } xs \text{ of } (\text{ [] } \rightarrow 0; (x': xs') \rightarrow (2 \ast x') + \text{sum } (g \text{ xs'}))
\]

We notice that unlike in previous examples, \text{sum} (\text{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, \text{sum} (\text{g } xs'), while keeping the functions already seen in mind. We inline the body of \text{sum}, perform the substitution of its arguments and inline the body of \text{g}:

\[
\text{case } (\text{ case } xs' \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x'': xs'') & \rightarrow (3 \ast x'') \text{ : f } xs''
\end{cases}) \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x': xs') & \rightarrow x' + \text{sum } xs'
\end{cases}
\]

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

\[
\text{case } xs'' \text{ of }
\begin{cases}
\text{ [] } & \rightarrow \text{ [] } \\
(x'': xs'') & \rightarrow (3 \ast x'') + \text{sum } (f \hspace{1pt} xs'')
\end{cases}
\]

Expressions

\[
e, f \ ::= n \mid x \mid g \mid f \hspace{1pt} f \mid \lambda \hspace{1pt} x . e \mid k \hspace{1pt} \tau \mid e_1 \ast e_2
\]

\[
\text{Values}
\]

\[
v \ ::= n \mid \lambda \hspace{1pt} n \hspace{1pt} x \hspace{1pt} e \hspace{1pt} k \hspace{1pt} \tau
\]

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

\[
h_2 \hspace{1pt} xs = \text{case } xs \text{ of }
\begin{cases}
\text{ [] } & \rightarrow 0 \\
(x'': xs'') & \rightarrow (2 \ast x') + \text{case } xs' \text{ of }
\begin{cases}
\text{ [] } & \rightarrow 0 \\
(x'': xs'') & \rightarrow (3 \ast x'') + h_2 \hspace{1pt} xs''
\end{cases}
\end{cases}
\]

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

\[
\text{vecDot } xs \text{ ys} = \text{sum} (\text{zipWith } (+) \hspace{1pt} xs \hspace{1pt} ys)
\]

This is simplified by our positive supercompiler to:

\[
\text{vecDot } xs \text{ ys} = h_1 \hspace{1pt} xs \hspace{1pt} ys = \text{case } xs \text{ of }
\begin{cases}
(x': xs') & \rightarrow \text{case } ys \text{ of }
\begin{cases}
(y': ys') & \rightarrow x' \ast y' + h_1 \hspace{1pt} xs' \hspace{1pt} ys' \\
& \rightarrow 0
\end{cases}
\end{cases}
\]

The intermediate list between \text{sum} and \text{zipWith} is transformed away, and the complexity is reduced from \(2 \mid x \hspace{1pt} s + \mid y \hspace{1pt} s\) to \(\mid x \hspace{1pt} s + \mid y \hspace{1pt} s\) (since this is matrix multiplication \(\mid x \hspace{1pt} s = \mid y \hspace{1pt} s\)).

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 \(G\) of global definitions whose right-hand sides are all values.

The language contains integer values \(n\) and arithmetic operations \(\ast\), 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 \(\ast\).

We abbreviate a list of expressions \(e_1 \ldots e_n\) as \(\tau\), and a list of variables \(x_1 \ldots x_n\) as \(\pi\). All functions have a specific arity and all applications must be saturated; hence \(\lambda x . \text{map} (\lambda y . y + 1) \hspace{1pt} x\) is legal whereas \(\text{map} (\lambda y . y + 1)\) 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 \(G\). The intended operational semantics is given in Figure 4, where \(\pi' \tau' e'\) is the capture-free substitution of expressions \(\pi\) for variables \(\tau\) in \(e'\).

A reduction context \(\mathcal{E}\) is a term containing a single hole \([\ ]\), which indicates the next expression to be reduced. The expression \(\mathcal{E}(e)\) is the term obtained by replacing the hole in \(\mathcal{E}\) with \(e\).
We encode letrec as an application containing fix, where fix is defined as
\( \lambda e.\ e\ (\lambda e.\ \text{fix}\ e) \).

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
\( a \ ::= \ x \mid n \oplus a \mid a \oplus n \mid a \oplus a \mid a\ e \).
Figure 5. Driving algorithm

4.1 Application Rule

In the driving algorithm rule R3 and rule R9 refer to $D_{app}(\cdot)$, defined in Figure 7. $D_{app}(\cdot)$ can be inline 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 $\equiv$ for syntactical equivalence of expressions.

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 $\rho$; 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 $\leq$ 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 $e \leq f, f$ 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 $e_i \leq e_j$. 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 Glicht (1995), which we adjust slightly to fit our language.

Definition 4.1 (Uniform terms). Let $s$ range over the set $N \cup X \cup K \cup \{caseof, let, letrec, primop, lambda, apply\}$, and let $caseof(\tau), let(\tau), letrec(\tau, e), primop(\tau), lambda(e)$, and $apply(\tau)$ denote a case, let, recursive let, primitive operation, lambda abstraction or application for all subexpressions $\tau, e, \alpha, \beta$. The set of terms $T$ is the smallest set of arity respecting symbol applications $s(\tau)$.
\[D_{app}(g, \overline{\tau}, \overline{\rho}, \overline{\sigma}) = (h', \pi, \{h', \text{Nothing}\})\],

where \(\pi = \text{fr}(R(g, \overline{\tau}))\),

\[D_{app}(g, \overline{\tau}, \overline{\rho}, \overline{\sigma}) = (x', \{h', \text{Just } R(g, \overline{\tau})\})\],

where \(x'\) fresh

\[D_{app}(g, \overline{\tau}, \overline{\rho}, \overline{\sigma}) = \text{GEN}(R(g, \overline{\tau}), t)\]

\[D_{app}(g, \overline{\tau}, \overline{\rho}, \overline{\sigma}) = \text{GEN}(R(g, \overline{\tau}), \text{head}(W))\]

\((\text{letrec } h = \lambda x. x' \in h \overline{\tau}, \text{used})\),

\((x', \text{used})\),

\(\pi = \text{fr}(R(g, \overline{\tau}))\), \(\rho' = \rho \cup (h, R(g, \overline{\tau}))\), \(\text{h fresh}\),

\(W = \{e|(n, \text{Just } e) \in \text{used}, n == h\}\)

\(\text{found} = \{n|(n, \text{Nothing}) \in \text{used}, n == h\}\)

**Figure 7. Driving of applications**

\[D[\text{append } xs \times y] \quad (*)\]

(Put \((h_0, \text{append } xs \times y)\) in \(\rho\) and transform according to the rules of the algorithm)

\[\text{case } xs \text{ of }\]

\[\begin{cases} \text{[] } & \rightarrow \text{xs} \\
(\text{xs': xs'}) \rightarrow D[\text{append } xs' \times y] \end{cases}\]

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

\[D[\text{append } xs' \times y] = (\text{Generalize the expression with } \text{append } xs' \times y)\]

\[D[\text{append } xs' \times y] / x, D[\text{xs'} / y] / D[\text{append } x \times y] = \text{letrec } h_0 \text{ xs ys = case xs of }\]

\[\begin{cases} \text{[]} & \rightarrow \text{ys} \\
(\text{x': xs'}) \rightarrow x' : h_0 \text{ xs'} \times y \end{cases}\]

\(\text{in } h_0 \text{ xs xs}\)

**Figure 10. Example of upwards generalization**

8. If the ground term \(t_y\) 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 \(\neq 0\) of \(D_{app}()\)).

The second alternative of \(D_{app}()\) in combination with the fourth alternative \((W \neq 0)\) is useful when transforming function calls that have the same parameter appearing twice, for example \(\text{append } xs \times y\) 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 \(\text{reverse}\) with an accumulating parameter is shown in Figure 11 with the definition of \(\text{reverse}\) as:

\[\text{rev } xs \times y = \text{case } xs \text{ of }\]

\[\begin{cases} \text{[]} & \rightarrow \text{ys} \\
(\text{x': xs'}) \rightarrow \text{rev } xs' (x' : ys) \end{cases}\]
\[ D[\text{rev } x s \{\} ] \]

(Put \((h_0, \text{rev } x s \{\})\) in \(\rho\) and transform the program according to the rules of the algorithm)

\[
\text{case } x s \text{ of } \left\{ \begin{array}{l}
\text{Right } (x', x) \to D[\text{rev } x s' (x' : \{\})] \\
\text{Right } e \to \text{Just } e
\end{array} \right.
\]

(Focus on the second branch and recall that \(\rho\) contains \(\text{rev } x s \{\}\) so alternative 3 of \(D_{\text{app}}(\) is triggered and the expression is generalized)

\[ D[\text{rev } x s' (x' : \{\})] \]

(Generalize the expression with \(\text{rev } x s \{\}\))

\[
[D[\text{rev } x s' (x' : \{\})]/\text{rev } x s' zs]\]

(Not \((h_1, \text{rev } x s' zs)\) in \(\rho\) and transform according to the rules of the algorithm)

\[
= \text{letrec } h_1 \ldots xs ys = \text{case } x s \text{ of } \left\{ \begin{array}{l}
\text{Right } (x' : xs' ) \to h_1 xs' (x' : ys) \\
\text{Just } e \to \text{Just } e
\end{array} \right.
\]

(Putting the two parts together)

\[
\text{case } x s \text{ of } \left\{ \begin{array}{l}
\text{Right } (x' : xs' ) \to \text{letrec } h_1 xs' ys = \text{case } x s \text{ of } \left\{ \begin{array}{l}
\text{Right } (x' : xs' ) \to h_1 xs' (x' : ys) \\
\text{Just } e \to \text{Just } e
\end{array} \right.
\end{array} \right.
\]

\[
\begin{array}{c|c|c|c|c|c}
\hline
\text{c} & \text{f} & t_y & \theta_1 & \theta_2 \\
\hline
e & \leq & \text{Just } e & x & e/x & \text{Just } e/x \\
\text{Right } e & \leq & \text{Right } (e, e') & x & e/x & (e, e')/x \\
\text{fac } y & \leq & \text{fac } (y - 1) & x & y/x & (y - 1)/x \\
\hline
\end{array}
\]

**Figure 9.** Examples of the homeomorphic embedding and the msg

**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 \(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 \((S, \leq)\) be a quasi-order. \((S, \leq)\) is a well-quasi-order if \(\leq\) is reflexive and transitive.

**Definition 5.3.** Let \((S, \leq)\) be a well-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 \(\rho\) 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 symbols, then any infinite sequence \(t_1, t_2, \ldots\) of terms from the set \(S\) contains two terms \(t_i, t_j\) with \(i < j\) such that \(t_i \leq t_j\).

We need to show that the memoization list \(\rho\) only contains elements that were in the initial input program:

**Lemma 5.5.** The second component of the memoization list, \(\rho\), can only contain terms from the set \(T\).

**Definition 5.6.** The weight of a call to the driving algorithm is \(\left|D[e_1]_{R, \bar{G}, \rho}\right| = (N - |\rho|, |R(e_1)|, |e_1|)\).

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 \(R: D[e_1]_{R, \bar{G}, \rho} = e_2\) in Figure 5 and Figure 7 and each recursive application \(D[e_2]_{R, \bar{G}, \rho'}\) in \(e_1\), \(\left|D[e_2]_{R, \bar{G}, \rho'}\right| < \left|D[e_1]_{R, \bar{G}, \rho}\right|\).
Lemma 5.8 (Totality). For all well-typed expressions \( e, D[e] \rho, G, \rho \) is matched by a unique rule in Figure 5.

Proposition 5.9 (Termination). The driving algorithm \( D[\_] \) 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 \( \rho \) 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 \( e', e \unrhd e' \), if for all contexts \( C \) such that \( C[e], C[e'] \) are closed, if evaluation of \( C[e] \) terminates then does evaluation of \( C[e'] \).
- \( e \) is operationally equivalent to \( e', e \unrhd e' \), if \( e \unrhd e' \) and \( e' \unrhd e \).

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

Definition 5.11 (Improvement, Strong Improvement).

- \( e \) is improved by \( e', e \rhd e' \), 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 \( e', e \rhd e' \), if \( e \rhd e' \) and \( e \unrhd e' \).

We use \( e \rhd v \) to denote that \( e \) evaluates to \( v \) using \( k \) function calls, and any other reduction rule as many times as it needs, and \( e' \rhd v' \) 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, \( e \equiv e' \), if \( e \rhd e' \) and \( e \unrhd e' \).

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)).\( f \) if \( e_1 \rhd e_1' \) and \( e_2 \rhd e_2' \) then \( (e_1 \equiv e_2) \iff (e_1' \equiv e_2') \).

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

Theorem 5.14 (Sands). If \( e \equiv e' \), 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 \( \rho \) an environment such that

- the range of \( \rho \) contains only closed expressions, and
- \( fv(e) \cap \text{dom}(\rho) = \emptyset \)

then \( e \equiv \rho[D[e]][\_], G, \rho] \).

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 \( \rho(D[\_] let \_x = e \in f]\_ G, \rho) = \rho(D[R(e/x)\_ G, \rho]) \).

Evaluated in the input term will eventually yield a context \( E[\_] \) with a term: \( R(\_ let \_x = e \in f) \iff R(\_ let \_x = e \in f) \iff R(\_ let \_x = e \in f) \iff R(\_ e \in f) \iff E(\_ e \in f) \).

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 active environment. 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 RTDSC 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 Stilman (1987) and they give an algorithm of complexity \( O(\text{size}(e) \times \text{size}(f)) \) to decide whether \( e \subseteq f \).

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.
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
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 done by Burstall and Darlington’s (1977) informal class of fold/unfold transformations.

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 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 (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 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:

\[ \text{foldr } f c = \text{build } g \text{ if } c \]

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 clumping 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 nofb 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.

$$\begin{align*}
D[\text{let } x = e \text{ in } f]_{R,\varphi,\rho} &= D[R[[e/x]/f]]_{R,\varphi,\rho}, \\
\text{if } x \in \text{strict}(f) \text{ and } f \text{ linear w.r.t } x \\
D[R[[e/x]/f]]_{R,\varphi,\rho} &= v, \\
\text{if } D[[e]/f]_{\varphi,\rho} = v, \\
D[R[[e/x]/f]]_{R,\varphi,\rho} &= f', \\
\text{if } D[[f]/f]_{\varphi,\rho} = f', x \in \text{strict}(f') \text{ and } f' \text{ linear w.r.t } x \\
\text{let } x = D[[e]/\rho,\varphi] \text{ in } D[R[f]]_{\rho,\varphi,\rho}, \\
\text{otherwise}
\end{align*}$$

Figure 12. Extended Let-rule
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 (Svenningson 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 Svenningson 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:

\[ \text{destroy } g \ (\text{unfoldr } \psi e) = g \psi e \]

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 German Vidal for explaining narrowing to us.

References

J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The
P. Narendran and J. Stillman. On the Complexity of Homeomorphic Em-
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 Amster-
dam, 1996.
Minker, editor, Foundations of Deductive Databases and Logic Pro-
X. Leroy. The Objective Caml system: Documentation and user’s manual,
S. Marlow and P. Wadler. Deforestation for higher-order functions. In John
Lauchbury and Patrick M. Sansom, editors, Functional Programming,
19820-2.
PhD thesis, Department of Computing Science, University of Glasgow,
April 27 1995.
R. Milner, M. Tofte, R. Harper, and D. MacQueen. The Definition of
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 Embed-
ddings. Technical Report 87-8, Computer Science Department, State
J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The
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,
W. Partain. The nofib benchmark suite of haskell programs. In John
Lauchbury and Patrick M. Sansom, editors, Functional Programming,
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 meta-
SIGPLAN-SIGACT Symposium on Principles of Programming Lan-
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
M.H. Sørensen. Convergence of program transformers in the metric space
M.H. Sørensen and R. Glück. An algorithm of generalization in positive su-
percompilation. In J.W. Lloyd, editor, International Logic Programming
M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evalua-
tion, deforestation, supercompilation, and GPC. In D. Sannella, editor,
Programming Languages and Systems — ESOP ’94. 5th European Sym-
posium 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
J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like
D. Syne. The # programming language, Jun 2008. URL http://
research.microsoft.com/fsharp.
In Partial Evaluation and Semantics-Based Program Manipulation, New
Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991),
A. Takano and E. Meijer. Shortcut deforestation in calculational form. In
V.F. Turchin. A supercompiler system based on the language Refal. SIG-
V.F. Turchin. Semantic definitions in Refal and automatic production of
compiers. In N.D. Jones, editor, Semantics-Directed Compiler Gener-
ation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94),
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.
V.F. Turchin. The concept of a supercompiler. ACM Transactions on
P. Wadler. Deforestation: transforming programs to eliminate trees. The-