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

Peter A. Jonsson
Johan Nordlander

Luleå University of Technology
Research Report
Department of Computer Science and Electrical Engineering
Division of EISLAB
Positive Supercompilation
for a Higher Order Call-By-Value Language

Extended Proofs

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

\[(\lambda x. y) (3/z).\]

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

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

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

\[
\text{append } xs \ ys = \text{case } xs \ of \\
\hspace{1cm} \text{[] } \rightarrow \ ys \\
\hspace{1.5cm} (x : xs) \rightarrow x : \text{append } xs \ ys
\]

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

\[
f = \text{append (append } xs' ys' \text{) } zs'
\]

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

\[
f = \text{case } xs' \ of \\
\hspace{1cm} \text{[] } \rightarrow \ texttt{append } ys' zs' \\
\hspace{1.5cm} (x : xs) \rightarrow \ texttt{append (x : append } xs' ys' \texttt{) } zs'
\]

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

\[
\text{append (x : append } xs' ys') \ zs'
\]

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

\[
x : \text{append } xs' ys' \\
\hspace{1cm} x : \text{case } xs' \ of \\
\hspace{1.5cm} \text{[] } \rightarrow \ ys' \\
\hspace{2cm} (x' : xs') \rightarrow x' : \text{append } xs' ys'
\]

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

The end result from this transformation is:

\[
f = \text{case } xs' \ of \\
\hspace{1cm} \text{[] } \rightarrow \ h_1 ys' zs' \\
\hspace{1.5cm} (x : xs) \rightarrow \ h_1 (h_2 x xs ys') zs'
\]

\[
h_1 xs ys = \text{case } xs \ of \\
\hspace{1cm} \text{[] } \rightarrow \ ys \\
\hspace{1.5cm} (x' : xs') \rightarrow x' : h_1 xs' ys
\]

\[
h_2 x xs ys = x : \text{case } xs \ of \\
\hspace{1cm} \text{[] } \rightarrow \ ys \\
\hspace{1.5cm} (x' : xs') \rightarrow h_2 x' xs' ys
\]

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

However, doing the exact opposite — that is, carefully delaying transformation of arguments to a function past inlining of its body — actually leads to the same result as Wadler obtains after transforming \texttt{append (append } xs ys) \texttt{). 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 shift the focus to the proper subterm of terms and to ensure 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 \texttt{sum (map square} \texttt{ys)}\texttt{). The functions used in the examples are defined as:

\[
square x = x \ast x \\
\text{map } f \ x s = \text{case } xs \ of \\
\hspace{1cm} \text{[] } \rightarrow y s \\
\hspace{1.5cm} (x : xs) \rightarrow f x : \text{map } f \ x s
\]

\[
\text{sum } x s = \text{case } xs \ of \\
\hspace{1cm} \text{[] } \rightarrow 0 \\
\hspace{1.5cm} (x : xs) \rightarrow x + \text{sum } xs
\]

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

\[
\text{case } \text{map square } \textbf{ys} \text{ of } \\
\hspace{1cm} \text{[] } \rightarrow 0 \\
\hspace{1.5cm} (x' : xs') \rightarrow x' + \text{sum } xs'
\]

After inlining \texttt{map} and substituting the arguments into the body the result becomes:

\[
\text{case (case } \texttt{ys } \texttt{of } \\
\hspace{1cm} \text{[] } \rightarrow \texttt{[] } \\
\hspace{1.5cm} (x' : xs') \rightarrow \texttt{square } x' : \texttt{map square } xs' \texttt{) } \texttt{of } \\
\hspace{1cm} \text{[] } \rightarrow \texttt{[] } \\
\hspace{1.5cm} (x' : xs') \rightarrow x' + \text{sum } xs'
\]

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:

\[
\text{case } \texttt{ys } \texttt{of } \\
\hspace{1cm} \text{[] } \rightarrow \texttt{0 } \\
\hspace{1.5cm} (x' : xs') \rightarrow \texttt{square } x' + \text{sum (map square } xs')
\]

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

\[
h_0 \ y s = \text{case } \texttt{ys } \texttt{of } \\
\hspace{1cm} \text{[] } \rightarrow \texttt{0 } \\
\hspace{1.5cm} (x' : xs') \rightarrow x' * x' + h_0 \ x s'
\]

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

The work by Ohori and Sasano (2007) can not fuse two successive applications of the same function, nor mutually recursive functions. We show that our algorithm can handle these two cases. We need the following new function definitions:
\[ \text{mapsq } xs = \text{case } xs \text{ of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x':xs') \rightarrow (x' \ast x') : \text{mapsq } xs' \]
\[ f \text{ } xs = \text{case } xs \text{ of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x':xs') \rightarrow (2 \ast x') : g \text{ } xs' \]
\[ g \text{ } xs = \text{case } xs \text{ of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x':xs') \rightarrow (3 \ast x') : f \text{ } xs' \]

Transforming \text{mapsq} (\text{mapsq } xs) will inline the outer \text{mapsq}, substitute the argument in the function body and inline the inner call to \text{mapsq}:
\[ \text{case ( } \text{case } \text{xs of} \]
\[ \quad (x':xs') \rightarrow (x' \ast x') : \text{mapsq } xs' \text{ of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x':xs') \rightarrow (x' \ast x') : \text{mapsq } xs' \]

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:
\[ \text{case } \text{xs of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x':xs') \rightarrow (x' \ast x' \ast x' \ast x') : \text{mapsq } (\text{mapsq } xs') \]

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:
\[ h_1 \text{ } xs = \text{case } \text{xs of} \]
\[ \quad [] \rightarrow [] \]
\[ \quad (x':xs') \rightarrow (x' \ast x' \ast x' \ast x') : h_1 \text{ } xs' \]

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

We now move down the outer case into each branch, and perform reductions until we end up with:
\[ \text{case } \text{xs of} \]
\[ \quad [] \rightarrow 0; (x':xs') \rightarrow (2 \ast x') + \text{sum } (g \text{ } xs') \]

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} (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 g:
\[ \text{case ( } \text{case } \text{xs of} \]
\[ \quad (x':xs'') \rightarrow (3 \ast x'') : f \text{ } xs'' \text{ of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x':xs'') \rightarrow x' + \text{sum } xs' \]

We now move down the outer case into each branch, and perform reductions:
\[ \text{case } \text{xs' of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x'':xs'') \rightarrow (3 \ast x'') + \text{sum } (f \text{ } xs'') \]

\[ \text{Expressions} \]
\[ e, f ::= n | x | g | f \overline{e} | \lambda \overline{e} | k \overline{e} | e_1 \oplus e_2 \]
\[ | \text{case } e \text{ of } \{ p_i \rightarrow e_i \} | \text{let } x = f \text{ in } e \]
\[ \text{letrec } g = e \text{ in } e \]

\[ p ::= n | k \overline{e} \]

\[ \text{Values} \]
\[ v ::= n | \lambda \overline{e} | k \overline{e} \]

\[ \text{Figure 1. The language} \]

We notice a familiar expression in \text{sum} (\text{f } xs''), and fold when reaching it. Adding it all together gives a new function \( h_2 \):
\[ h_2 \text{ } xs = \text{case } \text{xs of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x':xs') \rightarrow (2 \ast x') + \text{case } xs' \text{ of} \]
\[ \quad [] \rightarrow 0 \]
\[ \quad (x'':xs'') \rightarrow (3 \ast x'') + h_2 \text{ } xs'' \]

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}:
\[ \text{vecDot } xs \text{ ys} = \text{sum } (\text{zipWith } (\ast) \text{ } xs \text{ ys}) \]

This is simplified by our positive supercompiler to:
\[ \text{vecDot } xs \text{ ys} = h_1 \text{ } xs \text{ ys} \]
\[ h_1 \text{ } xs \text{ ys} = \text{case } \text{xs of} \]
\[ \quad (x':xs') \rightarrow \text{case } \text{ys of} \]
\[ \quad (y':ys') \rightarrow x' \ast y' + h_1 \text{ } xs' \text{ ys'} \]
\[ \quad [] \rightarrow 0 \]

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

3. Language

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

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

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

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

We denote the free variables of an expression \( e \) by \( \text{fv}(e) \), as defined in Figure 2. Along the same lines we denote the function names in an expression \( e \) as \( \text{fn}(e) \), 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 \( \overline{e} \overline{e}' \) is the capture-free substitution of expressions \( \overline{e} \) for variables \( \overline{e} \) in \( e' \).

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

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

4. Higher Order Positive Supercompilation

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

\[ R ::= [] | R \circ e | E \circ R | e \circ | e \circ e \]

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

In principle, an expression \( e \) is strict with regards to a variable \( x \) if it eventually evaluates \( x \) in other words, if \( e \mapsto \ldots \mapsto \overline{E}(x) \).

Such information is not computable in general, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 6, where the strict variables of an expression \( e \) are defined as all free variables of \( e \) except those that only appear under a lambda or not inside all branches of a case.

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

\[ a ::= x | n \oplus a | a \oplus n | a \oplus a \]

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...
arguments of a primitive operation are annoying our algorithm will

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

4.1 Application Rule

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

\[
\begin{align*}
&D[n]_{\mathcal{R}, G, \rho} = \mathcal{R}(n) \quad (R1) \\
&D[x]_{\mathcal{R}, G, \rho} = \mathcal{R}(x) \quad (R2) \\
&D[g]_{\mathcal{R}, G, \rho} = D_{app}(g, \mathcal{R}, G, \rho) \quad (R3) \\
&D[k\mathcal{R}]_{\mathcal{R}, G, \rho} = k \cdot D[k\mathcal{R}]_{\mathcal{R}, G, \rho} \quad (R4) \\
&D[x\mathcal{R}]_{\mathcal{R}, G, \rho} = \mathcal{R}(x, D[x\mathcal{R}]_{\mathcal{R}, G, \rho}) \quad (R5) \\
&D[\lambda x.e]_{\mathcal{R}, G, \rho} = \lambda D[\lambda x.e]_{\mathcal{R}, G, \rho} \\
&D[(\pi_1 + n_2)]_{\mathcal{R}, G, \rho} = D[\mathcal{R}(n_1)]_{\mathcal{R}, G, \rho}, \text{ where } n = n_1 + n_2 \quad (R7) \\
&D[\epsilon_1 + \epsilon_2]_{\mathcal{R}, G, \rho} = D[\epsilon_1]_{\mathcal{R}, G, \rho} \cup D[\epsilon_2]_{\mathcal{R}, G, \rho}, \text{ if } \epsilon_1 + \epsilon_2 = \epsilon \quad (R8) \\
&D[\epsilon_1]_{\mathcal{R}, G, \rho} \cup D[\epsilon_2]_{\mathcal{R}, G, \rho}, \text{ if } \epsilon_1 = \epsilon \text{ or } \epsilon_1 = \epsilon \quad (R9) \\
&D[\epsilon_1]_{\mathcal{R}, G, \rho} \cup D[\epsilon_2]_{\mathcal{R}, G, \rho}, \text{ otherwise} \\
&D[v\mathcal{R}]_{\mathcal{R}, G, \rho} = \mathcal{R}(v, D[v\mathcal{R}]_{\mathcal{R}, G, \rho}) \\
&D[\pi f\mathcal{R}]_{\mathcal{R}, G, \rho} = \mathcal{R}(\pi f, D[\pi f\mathcal{R}]_{\mathcal{R}, G, \rho}) \quad (R10) \\
&D[f]_{\mathcal{R}, G, \rho} = \mathcal{R}(f, D[f]_{\mathcal{R}, G, \rho}) \quad (R11) \\
&D[let x = v in f]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x. D[let x = v in f]_{\mathcal{R}, G, \rho}) \quad (R12) \\
&D[let x = y in f]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x. D[let x = y in f]_{\mathcal{R}, G, \rho}) \quad (R13) \\
&D[let x = e in f]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x. D[let x = e in f]_{\mathcal{R}, G, \rho}) \quad (R14) \\
&D[letrec x = v in e\epsilon]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x. D[letrec x = v in e\epsilon]_{\mathcal{R}, G, \rho}) \\
&D[case x of \{ p_1 \rightarrow e_1 \}]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x. D[case x of \{ p_1 \rightarrow e_1 \}]_{\mathcal{R}, G, \rho}) \quad (R15) \\
&D[case \pi ] = \mathcal{R}(\pi, D[\pi e]_{\mathcal{R}, G, \rho}) \quad (R16) \\
&D[c]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x. D[c]_{\mathcal{R}, G, \rho}) \quad (R17) \\
&D[letrec x_1 = e_1 in e_2\mathcal{R}]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x_1. D[letrec x_1 = e_1 in e_2\mathcal{R}]_{\mathcal{R}, G, \rho}) \quad (R18) \\
&D[letrec x_1 = e_1 in e_2\mathcal{R}]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda x_1. D[letrec x_1 = e_1 in e_2\mathcal{R}]_{\mathcal{R}, G, \rho}) \quad (R19) \\
&D[case \epsilon_1 of \{ p_i \rightarrow e_i \}]_{\mathcal{R}, G, \rho} = \mathcal{R}(\lambda \epsilon_1. D[case \epsilon_1 of \{ p_i \rightarrow e_i \}]_{\mathcal{R}, G, \rho}) \quad (R20)
\end{align*}
\]

Figure 5. Driving algorithm

On the other hand - if the application occurs at the head of a case expression, we may choose an branch on basis of the
constructor and leave the arguments unevaluated in the hope of
finding fold opportunities further down the syntax tree (rule R17).
The argumentation is analogous for lambda abstractions: if there
is a surrounding context we perform a beta reduction, otherwise we
drive its body.

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

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

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

Figure 8. Generalization

Care needs to be taken to ensure that recursive functions are not
inlined forever. The driving algorithm keeps a record of previously
seen applications in the memoization list \( \rho \); whenever it detects an
expression that is equivalent (up to renaming of variables) to a pre-
vious 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 \( \preceq \) 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 \( e \preceq f \), \( f \) contains all
subterms of \( e \), possibly embedded in other terms. For any inﬁnite
sequence \( e_0, e_1, \ldots \) there exists \( i \) and \( j \) such that \( i < j \) and \( e_i \preceq e_j \).
This condition is sufﬁcient to ensure termination.

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

Definition 4.1 (Uniform terms). Let \( S \) range over the set \( N \cup X \cup \)
\( K \cup \{ \text{caseof}, \text{ let}, \text{ letrec}, \text{ primop}, \text{ lambda}, \text{ apply} \} \), and let
\( \text{caseof}(\pi), \text{ let}(\pi), \text{ letrec}(\pi, e), \text{ primop}(\pi), \text{ lambda}(e), \) and
\( \text{apply}(\pi) \) denote a case, let, recursive let, primitive operation,
lambda abstraction or application for all subexpressions \( \pi, e \) and
\[ D_{\text{app}}(g, \bar{e}) R \bar{\varphi}, \rho = \begin{cases} (h', \mathcal{T}, \{(h', \text{Nothing})\}), & \text{if } \exists (h', t) \in \rho \cdot t = R(g \bar{e}) \\ (x', \{(x', \text{Just } R(g \bar{e}))\}), & \text{if } \exists (h', t) \in \rho \cdot t \subseteq R(g \bar{e}) \end{cases} \]

where \( \mathcal{T} = f_v(R(g \bar{e})) \)

where \( x' \) fresh

\[ D_{\text{app}}(g, \bar{e}) R \bar{\varphi}, \rho = \begin{cases} \text{GEN}(R(g \bar{e}), t) & \text{if } \exists (h', t) \in \rho \cdot t \subseteq R(g \bar{e}) \\ \text{GEN}(R(g \bar{e}), \text{head}(W)) & \text{if } W \neq \emptyset \\ \text{letrec } h = \exists x. e' \in h \mathcal{T}, \text{used}, \langle e', \text{used} \rangle, & \text{if found } \neq \emptyset \\ \text{otherwise} \end{cases} \]

where \( (g = v) \in G, \langle e', \text{used} \rangle = D[R(v \bar{e})] \)

\( \mathcal{T} = f_v(R(g \bar{e})), \rho' = \rho \cup (h, R(g \bar{e})), h \) fresh,

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

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

\[ \text{Figure 7. Driving of applications} \]

\( \tau. \text{The set of terms } T \text{ is the smallest set of arity respecting symbol applications } s(\mathcal{T}). \)

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

\[ x \preceq y, \ n_1 \preceq n_2, \ e \preceq f_i \text{ for some } i \]

\[ e_1 \preceq f_i, \ldots, e_n \preceq f_n \]

\[ s(e_1, \ldots, e_n) \preceq s(f_1, \ldots, f_n) \]

Examples of the homeomorphic embedding are shown in Figure 9.

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

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

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

- An instance of a term \( e \) is a term of the form \( \theta e \) for some substitution \( \theta \).
- A generalization of two terms \( e \) and \( f \) is a triple \( (t_\theta, \theta_1, \theta_2) \), where \( \theta_1, \theta_2 \) are substitutions such that \( \theta_1 t_{\theta_2} = e \) and \( \theta_2 t_{\theta_1} = f \).
- A most specific generalization (msg) of two terms \( e \) and \( f \) is a generalization \( (t_\theta, \theta_1, \theta_2) \) such that for every other generalization \( (t'_\theta, \theta'_1, \theta'_2) \) of \( e \) and \( f \) it holds that \( t_{\theta_2} = \text{instance of } t'_{\theta_1} \).

We refer to \( t_{\theta_2} \) as the ground term. For background information and an algorithm to compute most specific generalizations, see Lassez et al. (1988). Figure 9 also contains examples of the msg.

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

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

\[ \text{split}(s(x_1, \ldots, x_n)) = (s(x_1, \ldots, x_n), [e_1/x_1, \ldots, e_n/x_n]) \]

with \( x_1, \ldots, x_n \) fresh.

Alternative 2 of \( D_{\text{app}}() \) is for upwards generalization, and alternatives 3 and 4 are for the downwards generalization. This is exemplified below. The generalization algorithm is shown in Figure 8.

If the ground term \( t_q \) is a variable the algorithm drives the output from split, otherwise it will drive the output from the msg.

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

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

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

\[ \text{rev } xx ys = \text{case } xx \text{ of } \]

\[ [\rightarrow ys, (x' : xx') \rightarrow \text{rev } xx' (x' : ys)] \]
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.

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 and the weight of the entire transformation is a triple that contains the maximum length of the memoization list $\rho$, the weight of the term being transformed and the weight of the current term in focus.

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

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

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

The following lemma tells us that the set of finite sequences over a well-quasi-ordered set is well-quasi-ordered, with one proof by Nash-Williams (1963):

**Lemma 5.4** (Higman’s lemma). If a set $S$ is well-quasi-ordered, then the set $S^*$ of finite sequences over $S$ is well-quasi-ordered.

The weight of the entire transformation is a triple that contains the maximum length of the memoization list $\rho$, the weight of the term being transformed and the weight of the current term in focus. Such an $N$ exists follows from the homeomorphic embedding is a well-quasi-order and Kruskal’s Tree Theorem (Dershowitz 1987):

**Theorem 5.5** (Kruskal’s Tree Theorem). If $S$ is a finite set of function symbols, then any infinite sequence $t_1, t_2, \ldots$ of terms from the set $S$ contains two terms $t_i$ and $t_j$ with $i < j$ such that $t_i \leq t_j$.

Proof (Similar to Dershowitz 1987). Collapse all integers to a single 0-ary constructor, and all variables to a different 0-ary constructor. Only applications of the same arity are considered equal and only applications of finite arity are included. 3-arity application is different from a 4-arity application.

Suppose the theorem were false. Let the infinite sequence $\bar{t} = t_1, t_2, \ldots$ of terms be a minimal counterexample, measured by the size of the $t_i$. By the minimality hypothesis, the set of proper subterms of the $t_i$ must be well-quasi-ordered, or else there would be a smaller counterexample $t_1, t_2, \ldots, t_{i-1}, s_1, s_2, \ldots$, where
\(s_1, s_2, \ldots\) is a counterexample of proper subterms, such that \(s_1\) is a subterm of some \(t_l\) and all \(s_i\) in the counterexample are subterms of one of \(t_1, t_2, \ldots, t_{l-1}\). (None of \(t_1, t_2, \ldots, t_{l-1}\) can embed any of \(s_1, s_2, \ldots\), since that would mean that \(t_l\) also embeds in some \(t_j, i < l \leq j\).

Since the set \(S\) of function symbols is well-qua-ordered by \(\geq\), there must exist an infinite subsequence \(\tau\) of \(I\), the root (outermost) symbols of which constitute a quasi-ascending chain under \(<\). (Any infinite sequence of elements of a well-qua-ordered set must contain an infinite chain of quasi-ascending elements.) Since the set of proper subterms is well-qua-ordered, it follows by Lemma 5.4 that the set of finite sequences consisting of the immediate subterms of the elements in \(r\) is also well-qua-ordered. But then there would have to be an embedding in \(I\) itself, in which case it would not be a counterexample. \(\square\)

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

**Lemma 5.6.** The second component of the memoization list, \(\rho\), can only contain terms from the set \(T\).

**Proof.** Integers and fresh variables are equal, up to \(\equiv\), to the already existing integers and variables. Our only concern are the rules that introduce new terms that are not in \(T\). By inspection of the rules it is clear that only rules R3/R9 introduces such new terms. Inspection of the RHS of \(P_{app}(h, g, \rho)\):

- \(h':\) No recursive application in the RHS.
- \(x':\) No recursive application in the RHS.
- \(G\in (\mathcal{R}(g, \tau), t)\): Inline the definition of \(G\in(\cdot)\), two possible outcomes depending on what \(t_n\) is:
  - Case \(t_n \neq x: [D([\text{null}],[g], [\text{null}]), g, \rho] \times [D([\text{null}],[g], [\text{null}]), g, \rho] \times [D([\text{null}],[g], [\text{null}]), g, \rho]
  - Case \(t_n = x: [D([\text{null}],[g], [\text{null}]), g, \rho] \times [D([\text{null}],[g], [\text{null}]), g, \rho]

No new terms are created in either case and the memoization list \(\rho\) is not extended.

- \(G\in(\mathcal{R}(g, \tau), \text{head}(W)):\) Inline the definition of \(G\in(\cdot)\), two possible outcomes depending on what \(t_n\) is:
  - Case \(t_n \neq x: [D([\text{null}],[g], [\text{null}]), g, \rho] \times [D([\text{null}],[g], [\text{null}]), g, \rho] \times [D([\text{null}],[g], [\text{null}]), g, \rho]
  - Case \(t_n = x: [D([\text{null}],[g], [\text{null}]), g, \rho] \times [D([\text{null}],[g], [\text{null}]), g, \rho]

No new terms are created in either case and the memoization list \(\rho\) is not extended.

- \(\text{letrec } h = \lambda x. [D([\text{null}],[g], [\text{null}]), g, \rho] \times [h, x]:\) The newly created term \(h\) is kept outside of the recursive call of the driving algorithm. The memoization list, \(\rho\), is extended with terms from \(T\).

- \(\overline{\mathcal{D}(\mathcal{R}(g, \tau))}:\) No new terms are created, and the memoization list, \(\rho\), is extended with terms from \(T\). \(\square\)

We will show that each step of the driving algorithm will reduce the weight of what is being transformed. The constant \(N\) in the weight is the maximum length of the sequence of terms that are not related to each other by the homeomorphic embedding.

**Corollary 5.7.** For a fix set of function symbols \(G\) it follows from Kruskal’s Tree Theorem 5.5 that no infinite chains of terms can appear, so there exists a largest length of these chains \(N\).

We define the weight of driving a term as:

**Definition 5.8.** The weight of a call to the driving algorithm is

\[|D[x]|_{\mathcal{R}(g, \rho)} = (N - |\rho|, |\lambda. e|, 1)\]

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.

**Lemma 5.9.** The order between two tuples \((m_1, n_2, n_3)\) and \((m_1, m_2, m_3)\) is:

- \((n_1, n_2, n_3) < (m_1, m_2, m_3)\) if \(n_1 < m_1\)
- \((n_1, n_2, n_3) < (m_1, m_2, m_3)\) if \(n_1 = m_1\) and \(n_2 < m_2\)
- \((n_1, n_2, n_3) < (m_1, m_2, m_3)\) if \(n_1 = m_1, n_2 = m_2\) and \(n_3 < m_3\)

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

**Lemma 5.10.** If \((t_1, \theta_1, \theta_2) = n_u(\epsilon_1, \epsilon_2)\) and \(t_2 \neq \lambda\), then \(|t_2| < |\epsilon_1|\) and \(\theta_2 < |\epsilon_1|\)

**Lemma 5.11.** If \((t, \sigma) = \text{split } e\), then \(|t| < |\epsilon|\) and \(\sigma < |\epsilon|\)

**Lemma 5.12.** For each rule \(R: \mathcal{D}[\epsilon]\mathcal{R}(g, \rho)\) as \(\epsilon_1\) in Figure 5 and Figure 7 and each recursive application \(\mathcal{D}[\epsilon']\mathcal{R}(g, \rho)\) in \(\epsilon_1\), \(|\mathcal{D}[\epsilon']\mathcal{R}(g, \rho)| < |\mathcal{D}[\epsilon]\mathcal{R}(g, \rho)|\)

**Proof.** Inspection of rules:

- **R1** We have \(\mathcal{D}[\text{n}\mathcal{R}(g, \rho)\), and no recursive applications in the RHS, so Lemma 5.12 holds vacuously.
- **R2** We have \(\mathcal{D}[\text{n}\mathcal{R}(g, \rho)\), and no recursive applications in the RHS, so Lemma 5.12 holds vacuously.
- **R3** Argument analogous to R9.
- **R4** We have \(\mathcal{D}[\text{n}\mathcal{R}(g, \rho)\), and recursive applications \(\mathcal{D}[\epsilon']\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
- **R5** We have \(\mathcal{D}[\text{n}\mathcal{R}(g, \rho)\), and recursive applications \(\mathcal{D}[\epsilon']\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
- **R6** We have \(\mathcal{D}[\text{n}\mathcal{R}(g, \rho)\), and recursive applications \(\mathcal{D}[\epsilon']\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
- **R7** We have \(\mathcal{D}[\text{n}\mathcal{R}(g, \rho)\), and recursive application \(\mathcal{D}[\epsilon']\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
- **R8** a) \(\epsilon_1 + \epsilon_2 = \alpha: \) We have \(\mathcal{D}[\epsilon_1 + \epsilon_2]\mathcal{R}(g, \rho)\), and recursive applications \(\mathcal{D}[\epsilon_1 + \epsilon_2]\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
- **R9** b) \(\epsilon_1 = n \in \mathcal{R}(g, \rho)\), and recursive application \(\mathcal{D}[\epsilon_2]\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
- **R10** c) otherwise: We have \(\mathcal{D}[\epsilon_1 + \epsilon_2]\mathcal{R}(g, \rho)\), and recursive application \(\mathcal{D}[\epsilon_1 + \epsilon_2]\mathcal{R}(g, \rho)\) for each \(\epsilon'\) in \(\tau\).
We have LHS $D[g;\pi],[\rho,\gamma]$ and no recursive applications on the RHS, so Lemma 5.12 holds vacuously.

b) We have LHS $D[g;\pi],[\rho,\gamma]$ and no recursive applications on the RHS, so Lemma 5.12 holds vacuously.

c) We have LHS $D[g;\pi],[\rho,\gamma]$ and RHS $GEN(g;\gamma,t)$. In line the definition of $GEN(t)$ and get two possible outcomes depending on $t_0$:

Case $t_0 = x$:
$D[t_0;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$ and $D[t_0;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Lemma 5.10 $|\rho| < |R(g;\pi)|$ and $|\rho| < |R(g;\pi)|$, so
$(N - |\rho|, [R(g;\pi),\pi])] < (N - |\rho|, [R(g;\pi),\pi])$ and
$(N - |\rho|, [R(g;\pi),\pi])] < (N - |\rho|, [R(g;\pi),\pi])$.

Case $t_0 = \pi$:
$D[\pi;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$ and $D[\pi;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Lemma 5.11 $|\rho| < |R(g;\pi)|$ and $|\rho| < |R(g;\pi)|$, so
$(N - |\rho|, [R(g;\pi),\pi])] < (N - |\rho|, [R(g;\pi),\pi])$ and
$(N - |\rho|, [R(g;\pi),\pi])] < (N - |\rho|, [R(g;\pi),\pi])$.

d) Similar to $c$.

e) We have LHS $D[g;\pi],[\rho,\gamma]$ and recursive application
$D[\pi;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$ and
$D[\pi;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
The length of $\rho$ is one element longer than the length of $\rho$ and by definition 5.9 $(N - |\rho|, [R(g;\pi),\pi])] < (N - |\rho|, [R(g;\pi),\pi])]$.

R10 We have LHS $D[e;\pi],[\rho,\gamma]$ and recursive application
$D[\pi;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$ and
$D[\pi;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Definition 5.1 it follows that $e < |\pi| < \langle\lambda x.f\rangle$ and by Definition 5.9 $(N - |\rho|, [R(g;\pi),\pi])] < (N - |\rho|, [R(g;\pi),\pi])]$.

R11 We have LHS $D[e;\pi],[\rho,\gamma]$ and recursive application
$D[e;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$ and
$D[e;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Definition 5.1 it follows that $e < |\pi| < \langle\lambda x.f\rangle$.

R12 We have LHS $D[e;\pi],[\rho,\gamma]$ and recursive application
$D[e;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Definition 5.1 it follows that $e < |\pi| < \langle\lambda x.f\rangle$.

R13 We have LHS $D[e;\pi],[\rho,\gamma]$ and recursive application
$D[e;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Definition 5.1 it follows that $e < |\pi| < \langle\lambda x.f\rangle$.

R14 We have LHS $D[e;\pi],[\rho,\gamma]$ and recursive application
$D[e;\pi],[\rho,\gamma] = (N - |\rho|, [R(g;\pi),\pi])]$.
By Definition 5.1 it follows that $e < |\pi| < \langle\lambda x.f\rangle$.

The driving algorithm must be well defined for all inputs, hence a lemma about totality of the algorithm is needed. The proof is unchanged from the first author’s licentiate thesis (Jonsson 2008), and therefore not repeated here.

Lemma 5.13 (Totality). For all well-typed expressions $e$, $D[e;\pi],[\rho,\gamma]$ is matched by a unique rule in Figure 5.

With all these technical details in place, we can finally state our proposition for termination of the driving algorithm:

Proposition 5.14 (Termination). The driving algorithm $D[[\_]]$ terminates for all well-typed inputs.
Proof. Corollary 5.7 states that the weight of the transformation is defined. Lemma 5.6 guarantees that the memoization list $\rho$ only contains terms from the initial input. By Lemma 5.12 the weight of the transformation decreases for each step and by Lemma 5.13 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 5.15** (Operational Approximation and Equivalence).
- $e$ operationally approximates $e'$, $e \sqsubseteq e'$, if for all contexts $C$ such that $C[e], C[e']$ are closed, if evaluation of $C[e]$ terminates then so does evaluation of $C[e']$.
- $e$ is operationally equivalent to $e'$, $e \equiv e'$, if $e \sqsubseteq e'$ and $e' \sqsubseteq e$.

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

**Definition 5.16** (Improvement, Strong Improvement).
- $e$ is improved by $e'$, $e \sqsupset e'$, 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 $e'$, $e \triangleright e'$, iff $e \sqsupset e'$ and $e \equiv e'$.

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

To state the Improvement Theorem we view a transformation as the introduction of some new functions from a given set of definitions. We let $\{g_i\}_{i \in I}$ be a set of functions indexed by some set $I$, given by some definitions

$$\{g_i = \lambda x_1 \ldots x_{ni} e_i\}_{i \in I}$$

and let $\{e'_i\}_{i \in I}$ be a set of expressions such that for each $i \in I, f v(e'_i) \subseteq \{x_1 \ldots x_{ni}\}$. The following results relate to the transformation of the functions $g_i$ using the expressions $e'_i$: let $\{h_i\}_{i \in I}$ be a set of new functions given by the definitions

$$\{h_i = \frac{\overline{h_i}}{\overline{i}}\lambda x_1 \ldots x_{ni} e'_i\}_{i \in I}$$

**Theorem 5.17** (Sands Improvement theorem). If $g = e$ and $e \sqsupset C[g]$ then $g \sqsupset h$ where $h = C[h]$.

**Theorem 5.18** (Cost-equivalence theorem). If $e_i \sqsubseteq e'_i$ for all $i \in I$, then $g_i \sqsupset h_i$, $i \in I$.

We need a standard partial correctness result (Sands 1996a) associated with unfold-fold transformations

**Theorem 5.19** (Partial Correctness). If $e_i \equiv e'_i$ for all $i \in I$ then $h_i \sqsubseteq g_i$, $i \in I$.

which we combine with Theorem 5.17 to get total correctness for a transformation:

**Corollary 5.20.** If we have $e_i \sqsupset e'_i$ for all $i \in I$, then $g_i \sqsupset h_i$, $i \in I$.

Improvement theory in a call-by-value setting requires Sands operational metatheory for functional languages (Sands 1997), where the improvement theory is a simple corollary over the well-founded resource structure $\{n, 0, +, \geq\}$. For simplicity of presentation we instantiate the theorems by Sands to our language.

We borrow a set of improvement laws that will be useful for our proof:

**Lemma 5.21** (Sands (1996b), Improvement laws
1. If $e \sqsubseteq e'$ then $C[e] \sqsubseteq C[e']$.
2. If $e \equiv e'$ then $e \sqsubseteq e'$.
3. If $e \sqsubseteq e'$ and $e' \sqsubseteq e''$ then $e \sqsubseteq e''$.
4. If $e \mapsto e'$ then $e \sqsubseteq e'$.
5. If $e \sqsubseteq e'$ then $e \sqsubseteq e'$.

It is sometimes convenient to show that two expressions are related by showing that what they evaluate to is related.

**Lemma 5.22** (Sands (1996a)). If $e_1 \mapsto^r e_1'$ and $e_2 \mapsto^r e_2'$ then $(e_1 \sqsubseteq e_2 \implies e_1 \sqsubseteq e_2)$.

We need to show strong improvement for total correctness.

Since strong improvement is improvement in one direction and operational approximation in the other direction a set of approximation laws that correspond to the improvement laws in Lemma 5.21 is necessary.

**Lemma 5.23.** Approximation laws
1. If $e \sqsupset e'$ then $C[e] \sqsupset C[e']$.
2. If $e \equiv e'$ then $e \equiv e'$.
3. If $e \sqsupset e'$ and $e' \sqsupset e''$ then $e \sqsupset e''$.
4. If $e \mapsto e'$ then $e \sqsupset e'$.

Combining Lemma 5.21 and Lemma 5.23 gives us the final tools we need to prove strong improvement:

**Lemma 5.24.** Strong Improvement laws
1. If $e \sqsupset e'$ then $C[e] \sqsupset C[e']$.
2. If $e \equiv e'$ then $e \equiv e'$.
3. If $e \sqsupset e'$ and $e' \sqsupset e''$ then $e \sqsupset e''$.
4. If $e \mapsto e'$ then $e \sqsupset e'$.

A local form of the improvement theorem which deals with local expression-level recursion expressed with a fixed point combinator or with a letrec definition is necessary. This is analogous to the work by Sands (1996a), with slight modifications for call-by-value.

We need to relate local recursion expressed using fix, and the recursive definitions which the improvement theorem is defined for.

This is solved by a technical lemma that relates the cost of terms on a certain form to their recursive counterparts.

**Proposition 5.25.** For all expressions $e$, if $\lambda n.e$ is closed, then $\text{fix} (\lambda g.e) \sqsubseteq h$, where $h$ is a new function defined by $h = [\lambda n.h n/g] e$.

**Proof (Similar to Sands (1996a)).** Define a helper function $h^* = [\lambda n.\text{fix} (\lambda g.e) n/g] e$. Since $\text{fix} (\lambda g.e) \mapsto^r (\lambda f. (\lambda n.\text{fix} f n)) (\lambda g.e)$, we have $\text{fix} (\lambda g.e) \sqsubseteq h^*$. Since cost equivalence is a congruence relation we have that $[\lambda n.h n/g] e \sqsubseteq [\lambda n.\text{fix} (\lambda g.e) n/g] e$, and so by Theorem 5.18, we have a cost-equivalent transformation from $h^*$ to $h$, where $h = [h/h^*][\lambda n.h n/g] e = [\lambda n.h n/g] e$.

We state some simple properties that will be useful to prove our local improvement theorem

**Proposition 5.26.** Consequences of the letrec definition
i) \( \text{letrec } h = \lambda x. e \quad \text{ in } e' \hspace{1em} \iff \hspace{1em} [\lambda n. \text{fix } (h. \lambda x. e) n/h] e' \)

ii) \( \text{letrec } h = \lambda x. e \quad \text{ in } e' \hspace{1em} \iff \hspace{1em} \lambda n. \text{fix } (h. \lambda x. e) n \)

iii) \( \text{letrec } h = \lambda x. e \quad \text{ in } e' \hspace{1em} \iff \hspace{1em} [\text{letrec } h = \lambda x. e \quad \text{ in } h/h] e' \)

\textbf{Proof:} For i), expand the definition of \text{letrec} in the LHS, \((\lambda h. e') (\lambda n. \text{fix } (h. \lambda x. e) n)\) and evaluate it one step to \([\lambda n. \text{fix } (h. \lambda x. e) n/h] e'\). This is syntactically equivalent to the RHS, hence cost equivalent. For ii), set \(e' = h\) and perform the substitution from i). For iii), use the RHS of ii) in the substitution and notice it is equivalent to i).

This allows us to state the local version of the improvement theorem:

\textbf{Theorem 5.27 (Local improvement theorem).} If variables \(h, \pi\) include all the free variables of both \(e_0\) and \(e_1\), then

\[ \text{letrec } h = \lambda x. e_0 \quad \text{ in } e_0 \quad \vdash \quad \text{letrec } h = \lambda x. e_1 \quad \text{ in } e_1 \]

then for all expressions \(e\)

\[ \text{letrec } h = \lambda x. e_0 \quad \text{ in } e \quad \vdash \quad \text{letrec } h = \lambda x. e_1 \quad \text{ in } e \]

\textbf{Proof:} Define a new function \(g = [\lambda n. g/n] \lambda x. e_0\). By Proposition 5.25 \((\lambda h. g) [\lambda n. \text{fix } (h. \lambda x. e_0) n] \quad \vdash \quad \lambda n. \text{fix } (h. \lambda x. e_0) n/h \cdot e_1\)

Adding it all together yields \(\text{fix } (h. \lambda x. e_0) n/h \cdot e_1\). From transitivity and congruence properties of improvement we can deduce that \(\lambda n. \text{fix } (h. \lambda x. e_0) \quad \vdash \quad \lambda n. \text{fix } (h. \lambda x. e_1)\). By Proposition 5.26 we get \(\text{letrec } h = \lambda x. e_0 \vdash \lambda x. e_0 \quad \vdash \quad \text{letrec } h = \lambda x. e_1 \quad \text{ in } h\), which can be further expanded by congruence properties of improvement to \([\text{letrec } h = \lambda x. e_1 \quad \text{ in } h/h] e \quad \vdash \]

\[ [\text{letrec } h = \lambda x. e_0 \quad \text{ in } h/h] e \]

Using Proposition 5.26 one more time yields \(\text{letrec } h = \lambda x. e_0 \quad \text{ in } e \quad \vdash \quad \text{letrec } h = \lambda x. e_1 \quad \text{ in } e\) which proves our proposition.

\textbf{Proposition 5.28 (Total Correctness).} Let \(e\) be an expression, and \(\rho\) an environment such that

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

then \(e \quad \vdash \quad \rho(\text{let } e [\text{let } e \quad \text{ in } \rho])\).

The proof is in Appendix A.1 to Appendix A.20.

6. Benchmarks

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

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

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

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

The work on Supero by Mitchell and Runciman (2008) shows that there are open problems for supercompiling large Haskell programs. These problems are mainly relating to speed, both of the compiler, and of the transformed program. When they profiled Supero, they found that the majority of the time was spent in the homeomorphic embedding test. Our algorithm performs the test on a smaller part of the tree, so there is reason to believe that less time will be spent in the test for our algorithm. The complexity of the homeomorphic embedding has been investigated separately by Narendran and Stillman (1987) and they give an algorithm of complexity \(O(\text{size}(e) \times \text{size}(f))\) 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 increase 1316 bytes, on a binary of roughly 90k.

6.2 Factorial

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

6.3 Flip a Tree

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

6.4 Sum of Squares of a Tree

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

6.5 Kort’s Raytracer

The inner loop of a Raytracer (Kort 1996) is extracted and transformed. The total runtime, the number of allocations, the total size of allocations and the binary size are all decreased.
There are several ways the driving algorithm can be extended to make it more powerful. We show that with an extended Let-rule in combination with a disabled whistle for closed expressions we can evaluate many closed expressions.

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

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

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

\[
\text{bodyEx} = \begin{cases} \text{let } x = e \text{ in } & \begin{cases} \text{case } & \begin{cases} \rightarrow x; (x : xs) \rightarrow y \end{cases} \\ \rightarrow \mathcal{D}[\text{bodyEx}] \\ \rightarrow \mathcal{D}[\text{let } x = e \text{ in } & \begin{cases} \rightarrow x; (x : xs) \rightarrow y \end{cases} ] \end{cases} 
\end{cases}
\]

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

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

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 for the functional language Refal (Turchin 1989).

The positive supercompiler (Støersen 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 Støersen 2000). Negative information is the opposite of positive information, in-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time Before</th>
<th>Time After</th>
<th>Binary Size Before</th>
<th>Binary Size After</th>
<th>Allocations Before</th>
<th>Allocations After</th>
<th>Alloc Size Before</th>
<th>Alloc Size After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Append</td>
<td>105 844 704</td>
<td>89 820 912</td>
<td>89 484</td>
<td>90 800</td>
<td>2 70 035</td>
<td>180 032</td>
<td>2 160 280</td>
<td>1 440 256</td>
</tr>
<tr>
<td>Factorial</td>
<td>21 552</td>
<td>21 024</td>
<td>88 968</td>
<td>98 968</td>
<td>9</td>
<td>9</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>Flip a Tree</td>
<td>2 131 188</td>
<td>237 168</td>
<td>95 452</td>
<td>104 704</td>
<td>20 504</td>
<td>57</td>
<td>180 480</td>
<td>620</td>
</tr>
<tr>
<td>Sum of Squares of a Tree</td>
<td>276 102 012</td>
<td>28 737 648</td>
<td>95 452</td>
<td>104 912</td>
<td>4 194 338</td>
<td>91</td>
<td>29 360 406</td>
<td>908</td>
</tr>
<tr>
<td>Kort’s Raytracer</td>
<td>12 050 880</td>
<td>7 969 224</td>
<td>91 968</td>
<td>91 460</td>
<td>60 021</td>
<td>17</td>
<td>320 144</td>
<td>124</td>
</tr>
</tbody>
</table>

Table 1. Time, space and allocation measurements
equalities. 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 Nogu 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 straightforward to convert GPC to work on a call-by-value language, which makes it rather close to our work.

8.5 Other transformations

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

8.5.1 Partial Evaluation

Partial evaluation (Jones et al. 1993) is another instance of Burstall and Darlington’s (1977) informal class of fold/unfold transformations. If the partial evaluation is performed offline, the process is guided by program annotations that tells when to fold, unfold, instantiate and define. Binding-Time Analysis (BTA) is a a program analysis that annotates operations in the input program based on whether they are statically known or not. Partial evaluation does not remove intermediate structures, something we deem necessary to enable the programmer to write programs in the clear and concise listful style. Both deforestation and supercompilation simulate call-by-name evaluation in the transformer, whereas partial evaluation simulates call-by-value. It is suggested by Sørensen et al. (1994) that this might affect the strength of the transformation.

8.5.2 Short Cut Deforestation

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

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

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

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

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

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

8.5.3 Type-inference Based Short Cut Deforestation

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

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

The implications of the completeness is that a foldl 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

\[
D\left[\text{let } x = e \text{ in } f\right]_{\rho, g, \sigma} = \\
D[R(e/x)f]_{\rho, g, \sigma}, \\
D[R(v/x)f]_{\rho, g, \sigma}, \\
D[R(e/x)f']_{\rho, g, \sigma}, \\
\text{let } x = D[e]_{\rho, g, \sigma}, \text{ in } D[R(f)]_{\rho, g, \sigma},
\]

if \(x \in \text{strict}(f)\) and \(f\) linear w.r.t \(x\),

if \(D[e]_{\rho, g, \sigma} = v\),

if \(D[f]_{\rho, g, \sigma} = f', x \in \text{strict}(f')\) and \(f'\) linear w.r.t \(x\), otherwise

Figure 12. Extended Let-rule
the promising results from measurements suggests type-inference based short cut deforestation is a practical optimisation.

8.5.4 Zip fusion

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

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

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

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

9. Conclusions

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

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

9.1 Future work

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

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

Acknowledgments

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

A. Supporting Lemmas

We borrow a couple of technical lemmas from Sands (1996a), and adapt their proofs for call-by-value:

Lemma A.1 (Sands, p. 24). For all expressions e and value substitutions \( \theta \) such that \( h \notin \text{dom}(\theta) \), if \( e_0 \mapsto^1 e_1 \) then \( \text{letrec } h = \lambda x.e_1 \text{ in } [\theta(e_0)/z]e \mapsto^* \text{letrec } h = \lambda x.e_1 \text{ in } [h\theta(x)/z]e \)

The proof is in Figure 13.

Lemma A.2 (Sands, p. 25). \( \rho'(D[R(v \overline{x})]_{\bar{n}, \bar{\gamma}, \bar{\rho}'}) \mapsto^* \text{letrec } h = \lambda x.R_{\bar{\rho}}(v \overline{x}) \text{ in } \rho(D[R(v \overline{x})]_{\bar{n}, \bar{\gamma}, \bar{\rho}'}) \)

The proof is in Figure 14.

Lemma A.3. \( \mathcal{R}(\text{let } x = e \text{ in } f) \supseteq_s \text{let } x = e \text{ in } \mathcal{R}(f) \)

Proof. Notice that \( \mathcal{R}(\text{let } x = [] \text{ in } f) \) is a redex, and assume \( e \mapsto^k v \). The LHS evaluates in k steps \( \mathcal{R}(\text{let } x = e \text{ in } f) \mapsto^k \mathcal{R}(\text{let } x = v \text{ in } f) \mapsto \mathcal{R}([v/x]f) \), and the RHS evaluates in k steps \( \mathcal{R}(f) \mapsto^k \text{let } x = v \text{ in } \mathcal{R}(f) \mapsto [v/x]\mathcal{R}(f) \). Since contexts do not bind variables these two terms are equivalent and by Lemma 5.22 the initial terms are cost equivalent.

Lemma A.4. \( \mathcal{R}(\text{letrec } g = v \text{ in } e) \supseteq_s \text{letrec } g = v \text{ in } \mathcal{R}(e) \)

Proof. Translate both sides according to the definition of letrec into \( \mathcal{R}(R(\lambda g.e) (\lambda n.fix (\lambda g.v)n)) \supseteq_s (\lambda g.\mathcal{R}(e)) (\lambda n.fix (\lambda g.v)n) \).

Notice that \( \mathcal{R}([\cdot]) \) is a redex. The LHS evaluates in 0 steps \( \mathcal{R}(\lambda n.fix (\lambda g.v)n) \mapsto \mathcal{R}(([\lambda n.fix (\lambda g.v)n]/\bar{g})e) \) and the RHS evaluates in k steps \( \mathcal{R}(\mathcal{R}(e)) (\lambda n.fix (\lambda g.v)n) \mapsto [\lambda n.fix (\lambda g.v)n]/\bar{g}) \mathcal{R}(e) \). Since our contexts do not bind variables these two terms are equivalent and by Lemma 5.22 the initial terms are cost equivalent.

Lemma A.5. \( \mathcal{R}(\text{case } e \text{ of } \{ p_i \rightarrow e_i \}) \supseteq_s \text{case } e \text{ of } \{ p_i \rightarrow \mathcal{R}(e_i) \} \)

Proof. Notice that \( \mathcal{R}(\text{case } e \text{ of } \{ p_i \rightarrow e_i \}) \) is a redex, and assume \( e \mapsto^k n_j \). The LHS evaluates in k steps \( \mathcal{R}(\text{case } e \text{ of } \{ p_i \rightarrow e_i \}) \mapsto^k \mathcal{R}(\text{case } n_j \text{ of } \{ p_i \rightarrow e_i \}) \mapsto \mathcal{R}(e_i) \), and the RHS evaluates in k steps \( \text{case } e \text{ of } \{ p_i \rightarrow \mathcal{R}(e_i) \} \mapsto \text{case } n_j \text{ of } \{ p_i \rightarrow \mathcal{R}(e_i) \} \mathcal{R}(f) \mapsto \mathcal{R}(e_i) \). Since these two terms are equivalent the initial terms are cost equivalent by Lemma 5.22.

A.1 R1

We have that \( \rho(D[n]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = \rho(\mathcal{R}(\bar{n})) \), and the conditions of the proposition ensure that \( \mathcal{R}(\bar{n}) \cap \text{dom}(\rho) = \emptyset \), so \( \rho(\mathcal{R}(\bar{n})) = \mathcal{R}(\bar{n}) \). This is syntactically equivalent to the input, and we conclude \( \mathcal{R}(\bar{n}) \supseteq_s \rho(D[n]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \).

A.2 R2

We have that \( \rho(D[x]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = \rho(\mathcal{R}(\bar{x})) \), and the conditions of the proposition ensure that \( \mathcal{R}(\bar{x}) \cap \text{dom}(\rho) = \emptyset \), so \( \rho(\mathcal{R}(\bar{x})) = \mathcal{R}(\bar{x}) \). This is syntactically equivalent to the input, and we conclude \( \mathcal{R}(\bar{x}) \supseteq_s \rho(D[x]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \).

A.3 R3

Argument analogous to R9.

A.4 R4

We have that \( \rho(D[k\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = \rho(k\mathcal{D}[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \), and the conditions of the proposition ensure that \( \mathcal{D}[\bar{x}] \cap \text{dom}(\rho) = \emptyset \), so \( \rho(k\mathcal{D}[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = k\rho(D[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \). By the induction hypothesis, \( \quad \rho(D[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \), and by congruence properties of strong improvement (Lemma 5.24:1) \( k\rho(D[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \).

A.5 R5

We have that \( \rho(D[x]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = \rho(\mathcal{R}(x)) \), and the conditions of the proposition ensure that \( \mathcal{R}(x) \cap \text{dom}(\rho) = \emptyset \), so \( \rho(\mathcal{R}(x)) = \mathcal{R}(x) \). By the induction hypothesis, \( \quad \rho(D[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \), and by congruence properties of strong improvement (Lemma 5.24:1) \( \rho(D[\bar{x}]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \).

A.6 R6

We have that \( \rho(D[\lambda x.e]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = \rho(\lambda x.\mathcal{D}[e]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \), and the conditions of the proposition ensure that \( \mathcal{D}[e] \cap \text{dom}(\rho) = \emptyset \), so \( \rho(\lambda x.\mathcal{D}[e]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) = \lambda x.\rho(D[e]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \). By the induction hypothesis, \( \quad \rho(D[e]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \), and by congruence properties of strong improvement (Lemma 5.24:1) \( \lambda x.\rho(D[e]_{\bar{n}, \bar{\gamma}, \bar{\rho}}) \).
Proof. Expanding both sides according to the definition of letrec yields:

\[(\lambda h. \theta(e_0)/z)e \cdot (\lambda n. fix (\lambda h. \lambda x.x.e_1) n) \leq_{\theta} (\lambda h. [h \theta(\bar{x})]/z)e \cdot (\lambda n. fix (\lambda h. \lambda x.x.e_1) n)\]

and evaluating both sides one step \(\Rightarrow\) gives:

\[\left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta(e_0)/z]e \leq_{\theta} \left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta(\bar{x})]/z)e\]

From this we can see that it is sufficient to prove:

\[\left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta e_0] \leq_{\theta} \left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta(\bar{x})]\]

The substitution \(\theta\) can safely be moved out since \(\bar{x} \notin (\theta h) \cup ft(\lambda n. fix (\lambda h. \lambda x.x.e_1) n)\):

\[\left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta e_0] \leq_{\theta} \left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta(\bar{x})]\]

Performing evaluation steps on both sides yield:

\[\left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta e_0] \leq_{\theta} \left[\lambda n. fix (\lambda h. \lambda x.x.e_1) n\right]/h[\theta(\bar{x})]\]

The LHS and the RHS are cost equivalent, so by Lemma 5.22 the initial expressions are cost equivalent. \(\square\)

Figure 13. Proof of Lemma A.1

Proof (Similar to Sands (1996a)). By inspection of the rules for \(D[\cdot]\), all free occurrences of \(h\) in \(D[\bar{R}(\bar{v} \bar{x})][\cdot, \cdot, \cdot]\) must occur in subexpressions of the form \(h \bar{x}\). Suppose there are \(k\) such occurrences, which we can write as \(\theta_1 h \bar{x} \ldots \theta_k h \bar{x}\), where the \(\theta_i\) are just remappings of the variables \(\bar{x}\). So \(D[\bar{R}(\bar{v} \bar{x})][\cdot, \cdot, \cdot]\) can be written as \(\theta h \bar{x} \ldots \theta h \bar{x} \bar{x}/z_1 \ldots z_k\), where \(\bar{e}\) contains no free occurrences of \(h\). Then (substitution associates to the right):

\[
\rho'(D[\bar{R}(\bar{v} \bar{x})][\cdot, \cdot, \cdot]) = \left[\lambda \bar{x}. R(\bar{g} \bar{x})/h\right]\rho(D[\bar{R}(\bar{v} \bar{x})][\cdot, \cdot, \cdot])
\]

\[\leq_{\theta} \left[\lambda \bar{x}. R(\bar{g} \bar{x})/\rho[\theta_1 h \bar{x} \ldots \theta_k h \bar{x} \bar{x}/z_1 \ldots z_k]\right]\]

\[\leq_{\theta} \rho([\theta_1 R(\bar{g} \bar{x}) \ldots \theta_k R(\bar{g} \bar{x})/z_1 \ldots z_k]\bar{e})\]

\[\leq_{\theta} \text{(by Lemma A.1)}\]

\[\text{letrec } h = \lambda \bar{x}. R(\bar{v} \bar{x}) \mapsto \rho([\theta_1 h \bar{x} \ldots \theta_k h \bar{x} \bar{x}/z_1 \ldots z_k]\bar{e})\]

\[\equiv \text{letrec } h = \lambda \bar{x}. R(\bar{v} \bar{x}) \mapsto \rho(D[\bar{R}(\bar{v} \bar{x})][\cdot, \cdot, \cdot])\]

\(\square\)

Figure 14. Proof of Lemma A.2

A.7 R7

We have that \(\rho(D[n_1 \odot n_2]_{\cdot, \cdot, \cdot}) = \rho(D[R(n_1)]_{\cdot, \cdot, \cdot})\). By the induction hypothesis, \(R(n_1) \equiv_s \rho(D[R(n_1)]_{\cdot, \cdot, \cdot})\), and since \(R(n_1 \odot n_2) \Rightarrow R(n_2)\) it follows from Lemma 5.24:4 that \(R(n_1 \odot n_2) \equiv_s \rho(D[n_1 \odot n_2]_{\cdot, \cdot, \cdot})\).

A.8 R8

a) \(e_1 \odot e_2 = a\): We have that \(\rho(D[e_1 \odot e_2]_{\cdot, \cdot, \cdot}) = \rho(D[e_1]_{\cdot, \cdot, \cdot} \odot D[e_2]_{\cdot, \cdot, \cdot})\), and the conditions of the proposition ensure that \(f_1(R(e_1 \odot e_2)) \cap \text{dom}(\rho) = \emptyset\), so \(\rho(D[e_1]_{\cdot, \cdot, \cdot} \odot D[e_2]_{\cdot, \cdot, \cdot}) = \rho(D[e_1]_{\cdot, \cdot, \cdot} \cap \rho(D[e_2]_{\cdot, \cdot, \cdot}))\).

By the induction hypothesis \(e_1 \equiv_s \rho(D[e_1]_{\cdot, \cdot, \cdot})\) and \(e_2 \equiv_s \rho(D[e_2]_{\cdot, \cdot, \cdot})\), and by congruence properties of strong improvement (Lemma 5.24:1) \(R(e_1 \odot e_2) \equiv_s \rho(D[e_1 \odot e_2]_{\cdot, \cdot, \cdot})\).

b) \(e_1 = n\) or \(e_1 = a\): We have that \(\rho(D[e_1 \odot e_2]_{\cdot, \cdot, \cdot}) = \rho(D[e_1]_{\cdot, \cdot, \cdot} \odot D[e_2]_{\cdot, \cdot, \cdot})\) and \(R(e_1 \odot e_2) \equiv_s \rho(D[e_1]_{\cdot, \cdot, \cdot} \cap \rho(D[e_2]_{\cdot, \cdot, \cdot}))\).

By the induction hypothesis \(e_1 \equiv_s \rho(D[e_1]_{\cdot, \cdot, \cdot})\) and \(e_2 \equiv_s \rho(D[e_2]_{\cdot, \cdot, \cdot})\), and by congruence properties of strong improvement (Lemma 5.24:1) \(R(e_1 \odot e_2) \equiv_s \rho(D[e_1 \odot e_2]_{\cdot, \cdot, \cdot})\).

A.9 R9

A.9.1 Case: found in memolist

Suppose \(\exists h. \rho(h) \equiv \lambda x. R(\bar{g} v)\) and hence that \(D[R(\bar{g} v)]_\cdot \cdot \cdot = h \bar{x}\).

The conditions of the proposition ensure that \(\bar{x} \cap \text{dom}(\rho) = \emptyset\), so \(\rho(D[R(\bar{g} v)]_\cdot \cdot \cdot) = \rho(h \bar{x}) = (\lambda x. R(\bar{g} v)) \bar{x}\). However,
Suppose $\exists (h', t) \in \rho.t \leq \mathcal{R}(g \bar{v})$ and that $t = 0(\mathcal{R}(g \bar{v}))$, hence $D[\mathcal{R}(g \bar{v})][\rho, g, \mu] = x$.

The term on the RHS is discarded and replaced with a new term higher up in the tree, so it does not matter what the term is.

A.9.3 Case: Whistle blows #2
Suppose $\exists (h', t) \in \rho.t \leq \mathcal{R}(g \bar{v})$ and hence that $D[\mathcal{R}(g \bar{v})][\rho, g, \mu] = GEN(\mathcal{R}(g \bar{v}), t)$.

There are two subcases depending on the result of the msg of the two terms:

Case $t_{g} \neq x$: We have that $\rho(D[g \bar{v})][\rho, g, \mu] = \rho(D[t_{g}][\rho, g, \mu]D[t_{x}][\rho, g, \mu]) = \rho(D[t_{x}][\rho, g, \mu])$. By the induction hypothesis, $\rho \geq \rho$ and $t_{x} \geq t_{x}$ and by congruence properties of strong improvement (Lemma 5.24.1) $\mathcal{R}(g \bar{v}) \geq \rho(D[g \bar{v})][\rho, g, \mu]$.

Case $t_{g} = x$: We have that $\rho(D[g \bar{v})][\rho, g, \mu] = \rho(D[\sigma][\rho, g, \mu]D[t_{x}][\rho, g, \mu]) = \rho(D[\sigma][\rho, g, \mu])$. By the induction hypothesis, $\sigma \geq \sigma$ and $t_{x} \geq t_{x}$ and by congruence properties of strong improvement (Lemma 5.24.1) $\mathcal{R}(g \bar{v}) \geq \rho(D[g \bar{v})][\rho, g, \mu]$.

A.9.4 Case: W has members
Analogous to the previous case.

A.9.5 Case: Add to monolit
If $D[\mathcal{R}(g \bar{v})][\rho, g, \mu] = \rho(\text{letrec } h = \lambda x. \mathcal{R}(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v})$ where $\rho' = \rho \cup (h, \lambda x. \mathcal{R}(g \bar{v}))$ and $h \notin (\mathcal{T} \cup \text{dom}(\rho))$. We need to show that:

$\mathcal{R}(g \bar{v}) \geq \rho(\text{letrec } h = \lambda x. \mathcal{R}(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v})$.

Since $h, \bar{v} \notin \text{dom}(\rho)$ we have that $\rho(\text{letrec } h = \lambda x. \mathcal{R}(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v}) \equiv \text{letrec } h = \mathcal{R}(\rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v}) \equiv \mathcal{R}(\rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v})$. By Lemma A.1 we have that $\lambda x. \mathcal{R}(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \leq \mathcal{R}(g \bar{v})$. Since $h \notin \text{fv}(\rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu])$ this simplifies to $\mathcal{R}(g \bar{v}) \geq \text{letrec } h = \lambda x. \mathcal{R}(g \bar{v}) \text{ in } h \bar{v}$. It is necessary and sufficient to prove that:

\[
\exists \text{letrec } h = \lambda x. \mathcal{R}(g \bar{v}) \text{ in } h \bar{v} \geq \mathcal{R}(g \bar{v}) \geq \rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v}.
\]

By Theorem 5.27 it is sufficient to show:

\[
\exists \text{letrec } h = \lambda x. \mathcal{R}(g \bar{v}) \text{ in } h \bar{v} \geq \mathcal{R}(g \bar{v}) \geq \rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu]) \text{ in } h \bar{v}.
\]

A.9.6 Case: Inline
We have that $\rho(D[g \bar{v})][\rho, g, \mu] = \rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu])$. By the induction hypothesis $\mathcal{R}(v \bar{v}) \geq \rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu])$, and since $\mathcal{R}(g \bar{v}) \equiv \mathcal{R}(v \bar{v}) \Rightarrow \mathcal{R}(v \bar{v}) \geq \rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu])$ it follows from Lemma 5.24.4 that $\mathcal{R}(g \bar{v}) \geq \rho(D[\mathcal{R}(v \bar{v})][\rho, g, \mu])$.

A.10 R10
We have that $\rho(D[\mathcal{R}(\mathcal{X} f)][\rho, g, \mu]) = \rho(D[\mathcal{R}(\mathcal{X} f)][\rho, g, \mu])$. Evaluating the input term yields $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}([\mathcal{X} f])$, and evaluating the input to the recursive call yields $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}([\mathcal{X} f])$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 5.22 their ancestor terms are cost equivalent, $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$, and cost equivalence implies strong improvement. By the induction hypothesis $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$, therefore $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$.

A.11 R11
We have that $\rho(D[\mathcal{R}(\mathcal{X} f)][\rho, g, \mu]) = \rho(D[\mathcal{R}(\mathcal{X} f)][\rho, g, \mu])$. By the induction hypothesis $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$ it follows from Lemma 5.24.4.

A.12 R12
We have that $\rho(D[\mathcal{X} f][\rho, g, \mu]) = \rho(D[\mathcal{X} f][\rho, g, \mu])$. By the induction hypothesis $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$ it follows from Lemma 5.24.4 that $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$.

A.13 R13
We have that $\rho(D[\mathcal{X} f][\rho, g, \mu]) = \rho(D[\mathcal{X} f][\rho, g, \mu])$. By the induction hypothesis $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$ it follows that $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$.

A.14 R14
A.14.1 Case: $x \in \text{strict}(f)$
We have that $\rho(D[\mathcal{X} f][\rho, g, \mu]) = \rho(D[\mathcal{X} f][\rho, g, \mu])$. Evaluating the input term yields $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$.

A.14.2 Case: otherwise
We have that $\rho(D[\mathcal{X} f][\rho, g, \mu]) = \rho(D[\mathcal{X} f][\rho, g, \mu])$. By the induction hypothesis $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$ it follows that $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$.

A.15 R15
We have that $\rho(D[\mathcal{X} f][\rho, g, \mu]) = \rho(D[\mathcal{X} f][\rho, g, \mu])$. By the induction hypothesis $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$ it follows that $\mathcal{R}(\mathcal{X} f) \Rightarrow \mathcal{R}(\mathcal{X} f)$.
We have that $\rho [D \{ \text{case } x \text{ of } \{ p_i \rightarrow e_i \} \}]_{R, \varrho, \rho} = \rho [D \{ \text{case } x \text{ of } \{ p_i \rightarrow \mathcal{R}[e_i] \}]_{\varrho, \rho}$. And the conditions of the proposition ensure that for $\mathcal{S} = \{ \mathcal{S}[\text{case } x \text{ of } \{ p_i \rightarrow e_i \}] \} \cap \text{dom } \rho = \emptyset$, so $\rho [D \{ \text{case } x \text{ of } \{ p_i \rightarrow \mathcal{R}[e_i] \}]_{\varrho, \rho} = \mathcal{S}[\text{case } x \text{ of } \{ p_i \rightarrow \rho (\mathcal{R}[e_i]) \}]_{\varrho, \rho}$. By the induction hypothesis $\mathcal{S}[e_i] \subseteq \rho [D \{ \text{case } x \text{ of } \{ p_i \rightarrow e_i \} \}]_{\varrho, \rho}$. By Lemma A.5 the input is strongly improved by $\{ p_i \rightarrow \mathcal{R}[e_i] \}$, and therefore $\mathcal{S}[\text{case } x \text{ of } \{ p_i \rightarrow e_i \}]_{\varrho, \rho}$. 

References


