POSITIVE SUPERCOMPILATION
FOR A HIGHER-ORDER CALL-BY-VALUE LANGUAGE

PETER A. JONSSON AND JOHAN NORDLANDER

Luleå University of Technology, Department of Computer Science and Electrical Engineering
e-mail address: {pj,nordland}@csee.ltu.se

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

1. INTRODUCTION

Intermediate data structures such as lists allow functional programmers to write clear and concise programs, but carry a cost at run-time since additional heap cells need to be both allocated and garbage collected. Both deforestation [57] and supercompilation [47] are automatic program transformations which remove many of these intermediate structures. In a call-by-value context these transformations are unsound, however, as they might hide infinite recursion from the programmer. Consider the program

$$(\lambda x.y)\,(fac\,z).$$

This program could loop, if the value of z is negative. 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 non-termination in the original program has been removed, and hence the meaning of the program has been altered by the transformation.

This is unfortunate since 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 remain in memory during the computation.

Ohori and Sasano [35] 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. In this article we explore a different part of the design space: a more powerful transformation at the cost of some algorithmic complexity. The outcome is a meaning-preserving supercompiler for pure call-by-value

languages in general, together measurements from an implementation in a compiler for Timber [34], a pure call-by-value language.

Our current work is a necessary first step towards supercompiling impure call-by-value languages, of which there are many available today. Well-known examples are OCaml [25], Standard ML [29] and F# [49]. 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 easy to obtain 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. That is, eager specialization of function arguments risks destroying *fold* opportunities that might otherwise appear, something which may 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 transformation 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 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.

This article is an extended and improved version of a paper presented at POPL 2009 [20]. As well as clarifying a number of the examples and proofs, we give an improved formulation of $\mathcal{D}_{app}()$ presented in Section 4.1, and make a small change to how let-expressions are handled by the driving algorithm.

2. Examples

Wadler [57] uses the example *append* (*append xs ys*) *zs* and shows that his deforestation algorithm transforms the program so that it saves one traversal of the first list, thereby reducing the complexity from $2|xs| + |ys|$ to $|xs| + |ys|$.

If we naïvely change Wadler's algorithm to call-by-value semantics by eagerly attempting to transform arguments before attacking the body, we do not achieve this improvement. The definition for append is:

$$append\ xs\ ys = \textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ ys$$
$$(x' : xs')\ \rightarrow\ x' : append\ xs'\ ys$$

and we give an example of a hypothetical call-by-value variant of Wadler's deforestation algorithm that attacks arguments first:

$$append \; (append \; xs' \; ys') \; zs'$$

Inlining the body of the inner *append* and then pushing down the outer call into each branch gives

case xs' **of**
   $[] \; \rightarrow \; append \; ys' \; zs'$
   $(x : xs) \; \rightarrow \; append \; (x : append \; xs \; ys') \; zs'$

Transformation of the first branch will create a new function h_1 that is isomorphic to *append*, and call it. The second branch contains an embedding of the initial expression and blindly transforming it will lead to non-termination of the transformation algorithm. One must therefore split this expression in two parts: the subexpression $x : append \; xs \; ys'$ which we call h_2, and the outer expression $append \; z \; zs'$ where z is fresh. Continuing with $x : append \; xs \; ys'$ and inlining *append* gives

$x : $ **case** xs **of**
      $[] \; \rightarrow \; ys'$
      $(x' : xs') \; \rightarrow \; x' : append \; xs' \; ys'$

The second branch contains a renaming of the expression we named h_2, so we simply replace it with a call to h_2. Moving back to *append* $z \; zs'$ we notice that this expression is a renaming of the one called h_1, so we replace it with the call $h_1 \; z \; zs'$

Assembling the pieces gives us the final result:

letrec $h_1 \; xs \; ys \; = \;$ **case** xs **of**
                $[] \; \rightarrow \; ys$
                $(x' : xs') \; \rightarrow \; x' : h_1 \; xs' \; ys$
    $h_2 \; x \; xs \; ys \; = \; x : $ **case** xs **of**
                $[] \; \rightarrow \; ys$
                $(x' : xs') \; \rightarrow \; h_2 \; x' \; xs' \; ys$
in case xs' **of**
   $[] \; \rightarrow \; h_1 \; ys' \; zs'$
   $(x : xs) \; \rightarrow \; h_1 \; (h_2 \; x \; xs \; ys') \; zs'$

Notice that the intermediate structure from the input program is still there after the transformation, and the complexity is still $2|xs| + |ys|$! This can be compared to how the same example is transformed by Wadler's algorithm as shown in Figure 1. The reason our hypothetical call-by-value algorithm failed to improve the program is that it had to split expressions too early during the transformation, thereby preventing fold opportunities that occur in a call-by-name setting.

However, changing the call-by-value algorithm to do the exact opposite — that is, carefully delaying the transformation of arguments to a function past the inlining of its body, but only as far as strictness allows — actually leads to the same result that Wadler

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

Naming the first expression h_1 and inlining both occurrences of *append* gives

case (**case** xs' **of**
        $[] \rightarrow\ ys'$
        $(x_1 : xs_1)\ \rightarrow\ x_1 : append\ xs_1\ ys')$ **of**
  $[]\ \rightarrow zs'$
  $(x : xs)\ \rightarrow x : append\ xs\ zs'$

Pushing down the outer case-expression into both branches of the inner one and reducing the resulting case-expression of a known constructor leads to

case xs' **of**
    $[] \rightarrow$ **case** ys' **of**
          $[]\ \rightarrow zs'$
          $(x : xs)\ \rightarrow x : append\ xs\ zs'$
  $(x_1 : xs_1)\ \rightarrow\ x_1 : append\ (append\ xs_1\ ys')\ zs'$

Transform each branch separately. Transformation of the second branch in the first branch will create a new function h_2 that is isomorphic to *append*, and the second branch of the outer case is a renaming of our initial expression called h_1. Assembling all pieces yields the following result:

letrec $h_1\ xs\ ys\ zs\ =\ $ **case** xs **of**
                $[] \rightarrow$ **case** ys **of**
                      $[]\ \rightarrow\ zs$
                      $(y' : ys')\ \rightarrow\ y' : h_2\ ys'\ zs$
              $(x' : xs')\ \rightarrow\ x' : h_1\ xs'\ ys\ zs$
      $h_2\ xs\ ys\ =\ $ **case** xs **of**
                $[]\ \rightarrow\ ys$
                $(x' : xs')\ \rightarrow\ x' : h_2\ xs'\ ys$
  in $h_1\ xs'\ ys'\ zs'$

Figure 1: Wadler's algorithm transforming *append (append xs' ys') zs'*

obtains with *append (append xs ys) zs*. This is a key observation for obtaining deforestation under call-by-value without altering the semantics, and our transformation exploits it.

Except for the fundamental reliance on strictness analysis, which is necessary to preserve semantics, our transformation shares many of its rules with Wadler's algorithm. The transformation that is commonly referred to as case-of-case is crucial for our transformation, just like it is for a call-by-name algorithm. The case-of-case transformation is useful when a case-expression appears in the head of another case-expression, in which case the outer case context is duplicated and pushed into all branches of the inner case-expression. Our transformation also contains rules that correspond to ordinary evaluation which eliminate case-expressions that have a known constructor in their head or adds two primitive numbers. The mechanism that ensures termination basically looks for "similar" terms to

ones that have already been transformed, and if a similar term is encountered, the transformation will stop and split the term into smaller terms that are transformed separately. The remaining rules of our transformation simply shifts focus to the proper subexpression and ensures that the algorithm does not get stuck.

We claim that our transformation compares favorably with previous call-by-name transformations, and we now proceed with demonstrating the transformation on some common examples. The results of the transformation on these examples are identical to the results of Wadler's algorithm [57].

This does not hold in general, a counter-example is the transformation of the expression $zip\ (map\ f\ xs)\ (map\ g\ ys)$ where Wadler's algorithm will eliminate both intermediate structures and our transformation will only eliminate the first intermediate structure.

Our first example is transformation of $sum\ (map\ square\ ys)$, where the referenced functions are defined as:

$square\ x\ =\ x * x$
$map\ f\ xs = \textbf{case}\ xs\ \textbf{of}$
$\qquad\qquad [] \rightarrow\ ys$
$\qquad\qquad (x : xs) \rightarrow\ f\ x : map\ f\ xs$
$sum\ xs\quad = \textbf{case}\ xs\ \textbf{of}$
$\qquad\qquad [] \rightarrow\ 0$
$\qquad\qquad (x : xs) \rightarrow\ x\ +\ sum\ xs$

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

$\textbf{case}\ map\ square\ ys\ \textbf{of}$
$\quad [] \rightarrow\ 0$
$\quad (x' : xs') \rightarrow\ x'\ +\ sum\ xs'$

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

$\textbf{case}\ (\ \textbf{case}\ ys\ \textbf{of}$
$\qquad\qquad [] \rightarrow\ []$
$\qquad\qquad (x' : xs')\ \rightarrow\ (square\ x') : map\ square\ xs')\ \textbf{of}$
$\quad [] \rightarrow\ 0$
$\quad (x' : xs') \rightarrow\ x'\ +\ sum\ xs'$

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

$\textbf{case}\ ys\ \textbf{of}$
$\quad [] \rightarrow\ 0$
$\quad (x' : xs')\ \rightarrow\ square\ x'\ +\ sum\ (map\ square\ xs')$

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

$h_0\ ys =\ \textbf{case}\ ys\ \textbf{of}$
$\qquad\qquad [] \rightarrow\ 0$
$\qquad\qquad (x' : xs') \rightarrow\ x' * x'\ +\ h_0\ xs'$

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

The work by Ohori and Sasano [35] cannot fuse two successive applications of the same function, nor mutually recursive functions. We show in the next two examples that our transformation can handle these cases. We need the following new function definitions:

$$mapsq\ xs = \textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (x' * x') : mapsq\ xs'$$
$$f\ xs \qquad = \textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (2 * x') : g\ xs'$$
$$g\ xs \qquad = \textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (3 * x') : f\ xs'$$

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

$$\textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (x' * x') : mapsq\ xs')\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (x' * x') : mapsq\ xs'$$

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

$$\textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (x' * x' * x' * x') : mapsq\ (mapsq\ xs')$$

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

$$h_1\ xs = \textbf{case}\ xs\ \textbf{of}$$
$$[]\ \rightarrow\ []$$
$$(x' : xs')\ \rightarrow\ (x' * x' * x' * x') : h_1\ xs'$$

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

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

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

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

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

$$\textbf{case } (\ \textbf{case } xs' \textbf{ of}$$
$$[] \rightarrow []$$
$$(x'' : xs'') \rightarrow (3 * x'') : f\ xs'')\ \textbf{of}$$
$$[] \rightarrow 0$$
$$(x' : xs') \rightarrow x' + sum\ xs'$$

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

$$\textbf{case } xs' \textbf{ of}$$
$$[] \rightarrow 0$$
$$(x'' : xs'') \rightarrow (3 * x'') + sum\ (f\ xs'')$$

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

$$h_2\ xs = \textbf{case } xs \textbf{ of}$$
$$[] \rightarrow 0$$
$$(x' : xs') \rightarrow (2 * x') + \textbf{case } xs' \textbf{ of}$$
$$[] \rightarrow 0$$
$$(x'' : xs'') \rightarrow (3 * x'') + h_2\ xs''$$

The new function h_2 consumes a list and returns a number, so our algorithm has eliminated the intermediate list between f and sum.

Kort [22] studied a ray-tracer written in Haskell, and identified a critical function in the innermost loop of a matrix multiplication, called $vecDot$:

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

This is simplified by our positive supercompiler to:

$$vecDot\ xs\ ys = h_1\ xs\ ys$$
$$h_1\ xs\ ys = \textbf{case } xs \textbf{ of}$$
$$(x' : xs') \rightarrow \textbf{case } ys \textbf{ of}$$
$$(y' : ys') \rightarrow x' * y' + h_1\ xs'\ ys'$$
$$\_ \rightarrow 0$$
$$\_ \rightarrow 0$$

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

Expressions

$$e, f \quad ::= \quad n \mid x \mid g \mid f\,e \mid \lambda x.e \mid k\,\overline{e} \mid e_1 \oplus e_2 \mid \mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}$$
$$\quad\quad\quad\quad\quad \mid \quad \mathbf{let}\,x = f\,\mathbf{in}\,e \mid \mathbf{letrec}\,g = v\,\mathbf{in}\,e$$

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

Values

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

Figure 2: The language

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

Figure 3: Free variables of an expression

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 2.

Here we let variables and constructor symbols be denoted by x and k, respectively. The constructor symbols k range over a set K and we also assume that there is a separate set $\mathcal{G}$ of recursively defined function symbols, ranged over by g. In what follows we will assume that the meaning of such symbols is given by a recursive map G mapping symbols g to their defined value.

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

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

We denote the free variables of an expression e by $fv(e)$, as defined in Figure 3. Along the same lines we denote the function names in an expression e as $fn(e)$, defined in Figure 4.

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

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

Definition 3.1. Letrec is defined as:

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

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

Figure 4: Function names of an expression

Reduction contexts

$$
\mathcal{E} \quad ::= \quad \Box \;\mid\; \mathcal{E}\,e \;\mid\; (\lambda x.e)\,\mathcal{E} \;\mid\; k\,\overline{\mathcal{E}} \;\mid\; \mathcal{E} \oplus e \;\mid\; n \oplus \mathcal{E} \;\mid\; \mathbf{case}\,\mathcal{E}\,\mathbf{of}\,\{p_i \rightarrow e_i\} \;\mid\; \mathbf{let}\,x = \mathcal{E}\,\mathbf{in}\,e
$$

Evaluation relation

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

Figure 5: Reduction semantics

By defining letrec as syntactic sugar for other primitives we introduce an implicit requirement that the right hand side of letrec expressions must not contain any free variables except h. This is not a limitation since functions that contain free variables can be lambda lifted [17] to the top level.

A program is an expression with no free variables and all function names defined in G. The intended operational semantics is given in Figure 5, where $[\overline{e}/\overline{x}]e'$ is the capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e'.

A reduction context $\mathcal{E}$ is a term containing a single hole, $\Box$, which indicates the next expression to be reduced. The expression $\mathcal{E}\langle e \rangle$ is the term obtained by replacing the hole in $\mathcal{E}$ with e. $\overline{\mathcal{E}}$ denotes a list of terms with just a single hole, evaluated from left to right.

If a variable appears no more than once in a term, that term is said to be *linear* with respect to that variable. Like Wadler [57], we extend the definition slightly for linear case-expressions: no variable may appear in both the head and a branch, although a variable may appear in more than one branch. For example, the definition of *append* is linear is linear with respect to *ys*, although *ys* appears in both branches.

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 6. Three additional parameters appear as subscripts to the rewrite rules: a driving context $\mathcal{R}$, the set of global function definitions G and a memoization list ρ. The memoization list holds information about expressions already traversed and is explained more in detail in Section 4.1. The driving context $\mathcal{R}$ is smaller than $\mathcal{E}$, and is defined as follows:

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

Interestingly, this definition coincides with the evaluation contexts for a call-by-name language. The reason our transformation still preserves a call-by-value semantics is that beta reduction (rule R9) results in a let-binding, whose further specialization in rule R13 depends on whether the body expression f is strict in the bound variable x or not.

Our let-rule (R13) might change the order of computations, but since non-termination is commutative this does not matter in practice. Supercompiling impure languages requires stronger conditions for the let-rule, since expressions might contain effects other than non-termination. The difficulty of supercompiling an impure language is to find sufficient conditions that preserve soundness while still allowing the maximum amount of reordering of expressions.

In principle, an expression e is strict with regards to a variable x if evaluation of e eventually requires the value of x; in other words, if $e \mapsto \ldots \mapsto \mathcal{E}\langle x \rangle$. Such information is not computable in general, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 7, 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 the driving rules; i.e., all rules must be tried in the order they appear. Rule R10 is the default fallback case for applications and rule R19 is the default fallback case for case expressions. These rules extend the driving context $\mathcal{R}$ and zoom in on the next expression to be driven. The program is turned "inside-out" by moving the surrounding context $\mathcal{R}$ into all branches of the case-expression through rules R15 and R18. Rule R13 has a similar mechanism for let-expressions. Notice how the context is moved out of the recursive call in rule R5, whereas rule R7 recursively applies the driving algorithm to the full new term $\mathcal{R}\langle n \rangle$, forcing a re-traversal of the new term in search for for further reduction opportunities. Rule R12 is only allowed to match if the variable y is not freshly generated by the splitting mechanism described in the next section. Meta-variable a in rules R8 and R18 stands for an "annoying" expression; i.e., an expression that would be further reducible were it not for a free variable getting in the way. The grammar for annoying expressions is:

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

Some expressions should be handled differently depending on context. If a constructor application appears in an empty context, there is not much we can do but to drive the argument expressions (rule R4). On the other hand - if the application occurs at the head of a case-expression, we may choose a branch on basis of the constructor and leave the arguments unevaluated in the hope of finding fold opportunities further down the syntax tree (rule R16).

$$\mathcal{D}[\![n]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{R}\langle n\rangle \tag{R1}$$

$$\mathcal{D}[\![x]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{R}\langle x\rangle \tag{R2}$$

$$\mathcal{D}[\![g]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}_{app}(g)_{\mathcal{R},G,\rho} \tag{R3}$$

$$\mathcal{D}[\![k\,\overline{e}]\!]_{\square,G,\rho} \quad = \quad k\,\mathcal{D}[\![\overline{e}]\!]_{\square,G,\rho} \tag{R4}$$

$$\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{\square,G,\rho}\rangle \tag{R5}$$

$$\mathcal{D}[\![\lambda\overline{x}.e]\!]_{\square,G,\rho} \quad = \quad (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{\square,G,\rho}) \tag{R6}$$

$$\mathcal{D}[\![n_1\oplus n_2]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{\square,G,\rho},\ \text{where } n=n_1+n_2 \tag{R7}$$

$$\mathcal{D}[\![e_1\oplus e_2]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![e_1]\!]_{\square,G,\rho}\oplus\mathcal{D}[\![e_2]\!]_{\square,G,\rho},\ \text{if } e_1\oplus e_2=a \tag{R8}$$
$$\mathcal{D}[\![e_2]\!]_{\mathcal{R}\langle e_1\oplus\square\rangle,G,\rho},\ \text{if } e_1=n \text{ or } e_1=a$$
$$\mathcal{D}[\![e_1]\!]_{\mathcal{R}\langle\square\oplus e_2\rangle,G,\rho},\ \text{otherwise}$$

$$\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathbf{let}\ \overline{x}=\overline{e}\ \mathbf{in}\ f]\!]_{\mathcal{R},G,\rho} \tag{R9}$$

$$\mathcal{D}[\![e\,e']\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![e]\!]_{\mathcal{R}\langle\square\,e'\rangle,G,\rho} \tag{R10}$$

$$\mathcal{D}[\![\mathbf{let}\ x=n\ \mathbf{in}\ f]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle[n/x]f\rangle]\!]_{\square,G,\rho} \tag{R11}$$

$$\mathcal{D}[\![\mathbf{let}\ x=y\ \mathbf{in}\ f]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{\square,G,\rho},\ \text{if } y \text{ not freshly generated} \tag{R12}$$

$$\mathcal{D}[\![\mathbf{let}\ x=e\ \mathbf{in}\ f]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{\square,G,\rho},\ \text{if } x\in strict(f) \text{ and} \tag{R13}$$
$$x\in linear(f)$$
$$\mathbf{let}\ x=\mathcal{D}[\![e]\!]_{\square,G,\rho}\ \mathbf{in}\ \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{\square,G,\rho},\ \text{otherwise}$$

$$\mathcal{D}[\![\mathbf{letrec}\ g=v\ \mathbf{in}\ e]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{\square,G',\rho},\ \text{where } G'=G\cup(g,v) \tag{R14}$$

$$\mathcal{D}[\![\mathbf{case}\ x\ \mathbf{of}\ \{p_i\to e_i\}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathbf{case}\ x\ \mathbf{of}\ \{p_i\to\mathcal{D}[\![[p_i/x]\mathcal{R}\langle e_i\rangle]\!]_{\square,G,\rho}\} \tag{R15}$$

$$\mathcal{D}[\![\mathbf{case}\ k_j\,\overline{e}\ \mathbf{of}\ \{k_i\,\overline{x}_i\to e_i\}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\ \overline{x}_j=\overline{e}\ \mathbf{in}\ e_j\rangle]\!]_{\square,G,\rho} \tag{R16}$$

$$\mathcal{D}[\![\mathbf{case}\ n_j\ \mathbf{of}\ \{n_i\to e_i\}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{\square,G,\rho} \tag{R17}$$

$$\mathcal{D}[\![\mathbf{case}\ a\ \mathbf{of}\ \{p_i\to e_i\}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathbf{case}\ \mathcal{D}[\![a]\!]_{\square,G,\rho}\,\mathbf{of}\ \{p_i\to\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\square,G,\rho}\} \tag{R18}$$

$$\mathcal{D}[\![\mathbf{case}\ e\ \mathbf{of}\ \{p_i\to e_i\}]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{D}[\![e]\!]_{\mathcal{R}\langle\mathbf{case}\ \square\ \mathbf{of}\ \{p_i\to e_i\}\rangle,G,\rho} \tag{R19}$$

$$\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho} \quad = \quad \mathcal{R}\langle e\rangle \tag{R20}$$

Figure 6: Driving algorithm

$$strict(x) \qquad = \{x\}$$
$$strict(n) \qquad = \emptyset$$
$$strict(g) \qquad = \emptyset$$
$$strict(k\,\overline{e}) \qquad = strict(\overline{e})$$
$$strict(\lambda x.e) \qquad = \emptyset$$
$$strict(f\,e) \qquad = strict(f)\cup strict(e)$$
$$strict(\mathbf{let}\ x=e\ \mathbf{in}\ f) \qquad = strict(e)\cup(strict(f)\backslash\{x\})$$
$$strict(\mathbf{letrec}\ g=v\ \mathbf{in}\ f) \qquad = strict(f)$$
$$strict(\mathbf{case}\ e\ \mathbf{of}\ \{p_i\to e_i\}) = strict(e)\cup(\bigcap(strict(e_i)\backslash fv(p_i)))$$
$$strict(e_1\oplus e_2) \qquad = strict(e_1)\cup strict(e_2)$$

Figure 7: The strict variables of an expression

The argumentation is analogous for lambda abstractions: if there is a surrounding application context we perform a beta reduction, otherwise we proceed by driving the abstraction itself.

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

$$
\begin{aligned}
\mathcal{D}_{app}(g)_{\mathcal{R},G,\rho} &= h\,\overline{x} && \text{if } \exists(h,e_1)\in\rho\,.\,\sigma e_1 = \mathcal{R}\langle g\rangle && (1)\\
&\quad \text{where } \overline{x} = \sigma(fv(e_1)) \\
\mathcal{D}_{app}(g)_{\mathcal{R},G,\rho} &= \mathcal{R}\langle g\rangle && \text{if } \exists(h,e_1)\in\rho\,.\,e_1 \trianglelefteq \mathcal{R}\langle g\rangle \text{ and } \mathcal{R}\langle g\rangle \trianglelefteq e_1 && (2)\\
\mathcal{D}_{app}(g)_{\mathcal{R},G,\rho} &= [\mathcal{D}[\![\overline{f}]\!]_{\square,G,\rho}/\overline{y}]\mathcal{D}[\![f_g]\!]_{\square,G,\rho} && \text{if } \exists(h,e_1)\in\rho\,.\,e_1 \trianglelefteq \mathcal{R}\langle g\rangle && (3)\\
&\quad \text{where } (f_g,\overline{f},\overline{y}) = split(\mathcal{R}\langle g\rangle, e_1) \\
\mathcal{D}_{app}(g)_{\mathcal{R},G,\rho} &= [\mathcal{D}[\![\overline{f}]\!]_{\square,G,\rho}/\overline{y}]\mathcal{D}[\![f_g]\!]_{\square,G,\rho} && \text{if } \exists e_1 \in e\,.\,e_1 \trianglelefteq \mathcal{R}\langle g\rangle && (4a)\\
&\quad \textbf{letrec } h = \lambda\overline{x}.e \textbf{ in } h\,\overline{x} && \text{if } h \in fn(e) && (4b)\\
&\quad\; e && \text{otherwise} && (4c)\\
&\quad \text{where } (g,v) \in G, \\
&\qquad e = \mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}, \\
&\qquad \rho' = \rho \cup (h, \mathcal{R}\langle g\rangle), \\
&\qquad h \text{ fresh}, \\
&\qquad \overline{x} = fv(\mathcal{R}\langle g\rangle), \\
&\qquad (f_g,\overline{f},\overline{y}) = split(\mathcal{R}\langle g\rangle, e_1)
\end{aligned}
$$

Figure 8: Driving of applications

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 transformation 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 7. One might speculate whether the transformations thus missed will have adverse effects on the usefulness of our transformation in practice. We believe we have seen clear indications that this is not the case, and that the crucial factor is the ability to inline function bodies irrespective of whether arguments are values or not.

Our transformation always inlines functions unless the algorithm detects a risk of non-termination. Supero [30, Sec. 3.2] has a more advanced inlining strategy.

4.1. **Application Rule.** In the driving algorithm rule R3 refers to $\mathcal{D}_{app}()$, defined in Figure 8. $\mathcal{D}_{app}()$ can be inlined in the definition of the driving algorithm, it is merely given a separate name to improve the clarity of the presentation. Figure 8 contains some new notation: we use σ for a variable to variable substitution and $=$ for syntactic equivalence of expressions.

Care needs to be taken to ensure that recursive functions are not inlined forever. The driving algorithm keeps track of previously seen function applications in the memoization list ρ, which also associates a unique function name to each such expression. Whenever an expression that is equivalent up to renaming of variables to a previous application, a call to the associated function symbol is inserted instead. This is not sufficient to guarantee termination of the algorithm, but the mechanism is crucial for the complexity improvements mentioned in Section 2.

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

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

Figure 9: Examples of the homeomorphic embedding and the msg

In order to define the homeomorphic embedding we need a definition of uniform terms analogous to the one defined by Sørensen and Glück [45]. We slightly adjust their version to fit our language.

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

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

$$x \trianglelefteq y \qquad n_1 \trianglelefteq n_2 \qquad \frac{e \trianglelefteq f_i \text{ for some i}}{e \trianglelefteq s(f_1, \ldots, f_n)} \qquad \frac{e_1 \trianglelefteq f_1, \ldots, e_n \trianglelefteq f_n}{s(e_1, \ldots, e_n) \trianglelefteq s(f_1, \ldots, f_n)}$$

Whenever the whistle blows, our transformation 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 algorithm outlined by Sørensen [44], except that we need to reassemble the transformed subexpressions into a term of the original form instead of pulling them out as let-definitions, in order to preserve strictness. Our transformation is also more complicated because we perform the program extraction immediately, rather than constructing a large tree of terms 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 (*msg*).

Definition 4.3 (Most specific generalization).

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

For background information and an algorithm to compute most specific generalizations, see Lassez et al. [23]. Figure 9 contains examples of the homeomorphic embedding and the msg.

The most specific generalization is not always sufficient to split expressions. For expressions differing already in their roots, *msg* will return just a variable and substitutions equal to the input terms on that variable. If this happens we need to split expressions in a different way. We therefore define our function *split* using two alternatives; one that applies

when there is a non-trivial most specific generalization, and one that just splits along the spine of the first term in the other case.

Definition 4.4 (Split). For $t \in T$ we define $split(t_1, t_2)$ by:

$$
\begin{aligned}
split(s(\overline{e}_1), s'(\overline{e}_2)) &= (t_g, rng(\theta_1), dom(\theta_1)) \quad \text{if } s = s' \\
&= (s(\overline{x}), \overline{e}_1, \overline{x}) \qquad\qquad \text{otherwise}
\end{aligned}
$$

with $(t_g, \theta_1, \theta_2) = msg(s(\overline{e}_1), s'(\overline{e}_2))$ and $\overline{x}$ fresh.

Alternatives 2 and 4a of $\mathcal{D}_{app}()$ is for upwards generalization, and alternative 3 is for downwards generalization. This is exemplified below. All the examples of how our transformation works in Section 2 eventually terminate through a combination of alternative 1 and alternative 4b of $\mathcal{D}_{app}()$.

The second alternative of $\mathcal{D}_{app}()$ in combination with 4a is useful when transforming function calls that have the same parameter appearing twice, for example *append xs xs* as shown in Figure 10.

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, assuming the standard definition of reverse.

5. Correctness

The problem with using previous deforestation and supercompilation algorithms in a call-by-value context is that they might change the termination properties of programs. In this section we prove that our supercompiler both terminates itself, and preserves program termination behavior for all input.

5.1. **Termination.** In order to prove that the algorithm terminates we show that each recursive application of $\mathcal{D}[\![]\!]$ in the right-hand sides of Figure 6 and 8 has a strictly smaller weight than the left-hand side.

The weight of an expression is one plus the sum of the weight of its subexpressions, where variables, primitive numbers and function names have weight two. The weight of a fresh variable not in the initial input is one.

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

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

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

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 [32]:

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.*

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

(By rule 4 of $\mathcal{D}_{app}()$, put $(h_0,\ append\ xs\ xs)$ in ρ and transform according to the rules of the algorithm)

$=$ **case** xs **of**
$\quad\quad [] \rightarrow xs$
$\quad\quad (x' : xs') \rightarrow \mathcal{D}[\![x']\!]\ :\mathcal{D}[\![append\ xs'\ xs]\!]$

(Focus on $\mathcal{D}[\![\ append\ xs'\ xs]\!]$ and recall that ρ contains $append\ xs\ xs$ so alternative 2 of $\mathcal{D}_{app}(\ )$ is triggered and the transformation returns $append\ xs'\ xs$. This returns all the way up to the start (*) and the transformation continues there through alternative 4a)

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

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

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

$=$ $[xs/x,\ xs/y]$ **case** x **of**
$\quad\quad\quad [] \rightarrow y$
$\quad\quad\quad (x' : xs') \rightarrow \mathcal{D}[\![x']\!]\ :\mathcal{D}[\![append\ xs'\ y]\!]$

$=$ $[xs/x,\ xs/y]$ **case** x **of**
$\quad\quad\quad [] \rightarrow y$
$\quad\quad\quad (x' : xs') \rightarrow x'\ :h_0\ xs'\ y$

$=$ **letrec** $h_0\ xs\ ys\ =\ $ **case** xs **of**
$\quad\quad\quad\quad\quad\quad [] \rightarrow ys$
$\quad\quad\quad\quad\quad\quad (x' : xs') \rightarrow x' : h_0\ xs'\ ys$
$\quad$ **in** $h_0\ xs\ xs$

Figure 10: Example of upwards generalization

The weight of the entire transformation is a triple that contains the maximum length of the memoization list ρ denoted by N, the weight of the term being transformed and the weight of the current term in focus. That such an N exists follows from Kruskal's Tree Theorem [8] and the homeomorphic embedding relation being a well-quasi-order.

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 $t_i \trianglelefteq t_j$.*

Proof (Similar to Dershowitz [8]). Collapse all integers to a single 0-ary constructor, and all variables to a different 0-ary constructor.

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,

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

(By rule 4 of $\mathcal{D}_{app}()$, put $(h_0,\ rev\ xs\ [\,])$ in ρ and transform the program according to the rules of the algorithm)

case xs **of**
    $[\,]\ \rightarrow\ [\,]$
    $(x' : xs')\ \rightarrow\ \mathcal{D}[\![rev\ xs'\ (x' : [\,])]\!]$

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

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

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

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

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

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

$=\quad [(x' : [\,])/zs]\textbf{letrec}\ h_1\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of}$
                                     $[\,]\ \rightarrow\ ys$
                                     $(x' : xs')\ \rightarrow\ h_1\ xs'\ (x' : ys)$
              $\textbf{in}\ h_1\ xs'\ (x' : [\,])$

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

(Putting the two parts together)

case xs **of**
    $[\,]\ \rightarrow\ [\,]$
    $(x' : xs')\ \rightarrow\ \textbf{letrec}\ h_1\ xs\ ys\ =\ \textbf{case}\ xs\ \textbf{of}$
                                          $[\,]\ \rightarrow\ ys$
                                          $(x' : xs')\ \rightarrow\ h_1\ xs'\ (x' : ys)$
                   $\textbf{in}\ h_1\ xs'\ (x' : [\,])$

Figure 11: Example of downwards generalization

the set of proper subterms of the t_i must be well-quasi-ordered, or else there would be a smaller counterexample $t_1, t_2, \ldots, t_{l-1}, s_1, s_2, \ldots$, for some l such that s_1 is a subterm of t_1 and all $s_2, \ldots$ are subterms of one of $t_l, t_{l+1}, \ldots$. (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_i also is embedded in some $t_j, i < l \leq j$).

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

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 5.6. *Any infinite sequence $t_1, t_2, \ldots \in T^*$ contains two terms t_i and t_j with $i < j$ such that $t_i \trianglelefteq t_j$.*

Corollary 5.7. *There is a maximum N such that $t_1, t_2, \ldots, t_N \in T^*$ contains no terms t_i and t_j with $i < j$ and $t_i \trianglelefteq t_j$.*

We define the weight of driving a term as:

Definition 5.8. The weight of a call to the driving algorithm is $|\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho}| = (N - |\rho|, |\mathcal{R}\langle e \rangle|, |e|)$

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

Definition 5.9. The order between two tuples (n_1, n_2, n_3) and (m_1, m_2, m_3) is:

$$
\begin{array}{ll}
(n_1, n_2, n_3) < (m_1, m_2, m_3) & \text{if } n_1 < m_1 \\
(n_1, n_2, n_3) < (m_1, m_2, m_3) & \text{if } n_1 = m_1 \text{ and } n_2 < m_2 \\
(n_1, n_2, n_3) < (m_1, m_2, m_3) & \text{if } n_1 = m_1, \ n_2 = m_2 \text{ and } n_3 < m_3
\end{array}
$$

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

Lemma 5.10. *The second component of the memoization list ρ, can only contain terms from the set T.*

Proof. Integers and fresh variables are equal, up to $\trianglelefteq$, to the already existing integers and variables. Our only concern are the rules that introduce new terms that are not in T. The new function names h are the only new terms introduced by the algorithm. By inspection of the rules it is clear that only rule R3 introduces such new terms. Inspection of the RHS of $\mathcal{D}_{app}(),_{G,\rho}$:

  1: No recursive application in the RHS.
  2: No recursive application in the RHS.
  3: No new terms are created and the memoization list ρ is not extended.
  4a: No new terms are created and the memoization list ρ is not extended.
  4b: The newly created term $h\,\overline{x}$ is kept outside of the recursive call of the driving algorithm. The memoization list ρ, is extended with terms from T.

4c: No new terms are created, and the memoization list ρ, is extended with terms from T.

$\square$

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

Lemma 5.11. *For each rule $\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho} = e_1$ in Figure 6 and Figure 8 and each recursive application $\mathcal{D}[\![e']\!]_{\mathcal{R}',G,\rho'}$ in e_1, $|\mathcal{D}[\![e']\!]_{\mathcal{R}',G,\rho'}| < |\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho}|$*

Lemma 5.12 (Totality). *For all expressions $\mathcal{R}\langle e \rangle$, $\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho}$ is matched by a unique rule in Figure 6.*

Theorem 5.13 (Termination). *The driving algorithm $\mathcal{D}[\![\,]\!]$ terminates for all inputs.*

Proof. The weight of the transformation is defined because of Kruskal's Tree Theorem and the fact that the homeomorphic embedding is a well-quasi-order. Lemma 5.10 guarantees that the memoization list ρ only contains terms from the initial input. By Lemma 5.11 the weight of the transformation decreases for each step and by Lemma 5.12 we know that each recursive application will match a rule.

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

5.2. **Total Correctness.** The problem with previous deforestation and supercompilation algorithms in a call-by-value context is that they might change termination properties of programs. We prove that our supercompiler does not change what the program computes, nor does it alter whether a program terminates or not.

Sands [39] shows how a transformation can change the semantics in rather subtle ways – consider the function

$f\ x\ =\ x\ +\ 42$

It is clear that $f\ 0 \cong 42$ (where $\cong$ is semantic equivalence with respect to the current definition). Using this equality and replacing 42 in the function body with $f\ 0$ yields:

$f\ x\ =\ x\ +\ f\ 0$

This function will compute something entirely different than the original definition of f. We need some tools to ensure that the meaning of the original program is preserved and we therefore introduce the standard notions of operational approximation and equivalence. A general context C which is an expression with zero or more holes in the place of some subexpressions is used, and we say that an expression $C[e]$ is *closed* if there are no free variables in it.

Definition 5.14 (Operational Approximation and Equivalence).
- e operationally approximates e', $e \precsim e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed, if evaluation of $C[e]$ terminates then so does evaluation of $C[e']$.
- e is operationally equivalent to e', $e \cong e'$, if $e \precsim e'$ and $e' \precsim e$

The correctness of deforestation in a call-by-name setting has previously been shown by Sands [39] using his improvement theory. We use Sands's definitions for improvement and strong improvement:

Definition 5.15 (Improvement, Strong Improvement).

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

Note that improvement, $\trianglerighteq$, is not the same as the homeomorphic embedding, $\trianglelefteq$, defined previously.

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

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, where each function has a fixed arity α_i and are given by some definitions

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

and let $\{e_i'\}_{i \in I}$ be a set of expressions such that for each $i \in I$, $fv(e_i') \subseteq \{x_1 \ldots x_{\alpha_i}\}$. 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

$$\{h_i = [\overline{h}/\overline{g}]\lambda x_1 \ldots x_{\alpha_i}.e_i'\}_{i \in I}$$

Theorem 5.16 (Sands Improvement theorem). *If $g = e$ and $e \trianglerighteq C[g]$ then $g \trianglerighteq h$ where $h = C[h]$.*

Theorem 5.17 (Cost-equivalence theorem). *If $e_i \trianglelefteq\trianglerighteq e_i'$ for all $i \in I$, then $g_i \trianglelefteq\trianglerighteq h_i$, $i \in I$.*

We need a standard partial correctness result [39] associated with unfold-fold transformations

Theorem 5.18 (Partial Correctness). *If $e_i \cong e_i'$ for all $i \in I$ then $h_i \sqsubseteq_\approx g_i$, $i \in I$.*

which we combine with Theorem 5.16 to get total correctness for a transformation:

Corollary 5.19. *If we have $e_i \trianglerighteq_s e_i'$ for all $i \in I$, then $g_i \trianglerighteq_s h_i$, $i \in I$.*

Improvement theory in a call-by-value setting requires Sands operational metatheory for functional languages [41], where the improvement theory is a simple corollary over the well-founded resource structure $\langle \mathbb{N}, 0, +, \geq \rangle$. For simplicity of presentation we instantiate Sands's theorems to our language. We use $\equiv$ to denote expressions equal up to renaming of bound variables and borrow a set of improvement laws that will be useful for our proof:

Lemma 5.20 (Sands [40]). *Improvement laws*
(1) *If $e \trianglerighteq e'$ then $C[e] \trianglerighteq C[e']$.*
(2) *If $e \equiv e'$ then $e \trianglerighteq e'$.*
(3) *If $e \trianglerighteq e'$ and $e' \trianglerighteq e''$ then $e \trianglerighteq e''$*
(4) *If $e \mapsto e'$ then $e \trianglerighteq e'$.*
(5) *If $e \trianglerighteq e'$ then $e \sqsubseteq_\approx e'$.*

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

Lemma 5.21 (Sands [39]). *If $e_1 \mapsto^r e_1'$ and $e_2 \mapsto^r e_2'$ then $(e_1' \trianglelefteq\trianglerighteq e_2' \Leftrightarrow e_1 \trianglelefteq\trianglerighteq e_2)$.*

We need to show strong improvement in order to prove 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 5.20 is necessary.

Lemma 5.22. *Approximation laws*

(1) *If $e \sqsupseteq_{\approx} e'$ then $C[e] \sqsupseteq_{\approx} C[e']$.*
(2) *If $e \equiv e'$ then $e \sqsupseteq_{\approx} e'$.*
(3) *If $e \sqsupseteq_{\approx} e'$ and $e' \sqsupseteq_{\approx} e''$ then $e \sqsupseteq_{\approx} e''$*
(4) *If $e \mapsto e'$ then $e \sqsupseteq_{\approx} e'$.*

Combining Lemma 5.20 and Lemma 5.22 gives us the final tools we need to prove strong improvement:

Lemma 5.23. *Strong Improvement laws*

(1) *If $e \trianglerighteq_s e'$ then $C[e] \trianglerighteq_s C[e']$.*
(2) *If $e \equiv e'$ then $e \trianglerighteq_s e'$.*
(3) *If $e \trianglerighteq_s e'$ and $e' \trianglerighteq_s e''$ then $e \trianglerighteq_s e''$*
(4) *If $e \mapsto e'$ then $e \trianglerighteq_s e'$.*

If two expressions are improvements of each other, they are considered cost equivalent. Cost equivalence also implies strong improvement, which will be useful in many parts of our proof of total correctness for our supercompiler.

Definition 5.24 (Cost equivalence)**.** The expressions e and e' are cost equivalent, $e \trianglelefteq\trianglerighteq e'$ iff $e \trianglerighteq e'$ and $e' \trianglerighteq e$

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 [39], 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.

Theorem 5.25. *For all expressions e, if $\lambda g.e$ is closed, then $fix\,(\lambda g.e) \trianglelefteq\trianglerighteq h$, where h is a new function defined by $h = [\lambda n.h\,n/g]e$.*

Proof (Similar to Sands [39]). Define a helper function $h^- = [\lambda n.fix\,(\lambda g.e)\,n/g]e$. Since $fix\,(\lambda g.e) \mapsto^1 (\lambda f.f\,(\lambda n.fix\,f\,n))\,(\lambda g.e) \mapsto (\lambda g.e)\,(\lambda n.fix\,(\lambda g.e)\,n) \mapsto [\lambda n.fix\,(\lambda g.e)\,n/g]e$ and $h^- \mapsto^1 [\lambda n.fix\,(\lambda g.e)\,n/g]e$ it follows by Lemma 5.21 that $fix\,(\lambda g.e) \trianglelefteq\trianglerighteq h^-$. Since cost equivalence is a congruence relation we have that $[\lambda n.h^-\,n/g]e \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda g.e)\,n/g]e$, and so by Theorem 5.17, we have a cost-equivalent transformation from h^- to h, where $h = [h/h^-][\lambda n.h^-\,n/g]e = [\lambda n.h\,n/g]e$. $\qquad\square$

We state some simple properties that will be useful for proving our local improvement theorem

Theorem 5.26. *Consequences of the letrec definition*

  i): **letrec** $h = \lambda\overline{x}.e$ **in** $e' \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda\overline{x}.e)\,n/h]e'$
 ii): **letrec** $h = \lambda\overline{x}.e$ **in** $h \trianglelefteq\trianglerighteq \lambda n.fix\,(\lambda h.\lambda\overline{x}.e)\,n$
iii): **letrec** $h = \lambda\overline{x}.e$ **in** $e' \trianglelefteq\trianglerighteq [\textbf{letrec}\ h = \lambda\overline{x}.e\ \textbf{in}\ h/h]e'$

Proof. For i), expand the definition of letrec in the LHS, $(\lambda h.e')\,(\lambda n.fix\,(\lambda h.\lambda\overline{x}.e)\,n)$ and evaluate it one step to $[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e)\,n/h]e'$. 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). $\square$

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

Theorem 5.27 (Local improvement theorem). *If variables h and $\overline{x}$ include all the free variables of both e_0 and e_1, then if*

$$\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e_0 \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e_1$$

then for all expressions e

$$\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, e$$

Proof. Define a new function $g = [\lambda n.g\,n/h]\lambda\overline{x}.e_0$. By Proposition 5.25 $g \trianglelefteq\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_0)$. Use this, the congruence properties, and the properties listed in Proposition 5.26 to transform the premise of the theorem:

$$\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e_0 \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e_1$$

$$[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0)\,n/h]e_0 \trianglerighteq_s [\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0)\,n/h]e_1$$

$$\lambda\overline{x}.[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0)\,n/h]e_0 \trianglerighteq_s \lambda\overline{x}.[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0)\,n/h]e_1$$

$$[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0)\,n/h]\lambda\overline{x}.e_0 \trianglerighteq_s [\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0)\,n/h]\lambda\overline{x}.e_1$$

$$[\lambda n.g\,n/h]\lambda\overline{x}.e_0 \trianglerighteq_s [\lambda n.g\,n/h]\lambda\overline{x}.e_1$$

So by Corollary 5.19, $g \trianglerighteq_s g'$ where $g' = [g'/g][\lambda n.g\,n/h]\lambda\overline{x}.e_1 = [\lambda n.g\,n/h]\lambda\overline{x}.e_1$. Hence by Proposition 5.25, $g' \trianglelefteq\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_1)$. Adding it all together yields $fix\,(\lambda h.\lambda\overline{x}.e_0) \trianglelefteq\trianglerighteq g \trianglerighteq_s g' \trianglelefteq\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_1)$. From the transitivity and congruence properties of improvement we can deduce that $\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_0) \trianglerighteq_s \lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)$. By Proposition 5.26 we get $\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, h \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, h$, which can be further expanded by congruency properties of improvement to $[\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, h/h]e \trianglerighteq_s [\textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, h/h]e$. Using Proposition 5.26 one more time yields $\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, e$, which proves our theorem. $\square$

This allows us to state the total correctness theorem for our transformation:

Theorem 5.28 (Total Correctness). *Let $\mathcal{R}\langle e\rangle$ be an expression, G a recursive map, and ρ an environment such that*

- *the range of ρ contains only closed expressions, and*
- *$fv(\mathcal{R}\langle e\rangle) \cap dom(\rho) = \emptyset$, and*

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

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

6. Benchmarks

In this section we provide measurements on a set of common examples from the litera-ture on deforestation and perform a detailed analysis for each example. We show that our positive supercompiler removes intermediate structures and can improve the performance by an order of magnitude for certain benchmarks. The supercompiler was implemented as a pass in the Timber compiler [34]. Timber is a pure functional call-by-value language which is very close to the language we describe in Section 3, and for the scope of this article it can be thought of as a strict variant of Haskell. We have left out the full details of the instrumentation of the run-time system but it is available in a separate report [19].

All measurements were performed on an idle machine running in an *xterm* terminal environment. Each test was run 10 consecutive times and the best result was selected because the programs are deterministic and the best result must appear under the minimum of other activity. The number of allocations and the total allocation sizes remained constant over all runs.

Raw data for the time and size measurements before and after supercompilation are shown in Table 1, and allocation measures in Table 2. Compilation times are shown in Table 3. The time column is the number of clock ticks obtained from the *RDTSC* instruction available on Intel/AMD processors, and the binary size is in bytes. The total number of allocations and the total memory size in bytes allocated by the program are displayed in their respective column. The compilation times are measured in seconds and times from left to right are for producing an object file, producing an executable binary, and the corresponding operations with supercompilation turned on.

Binary sizes are slightly increased by the supercompiler, but all run-times are faster. The main reason for the performance improvement is the removal of intermediate structures, reducing the number of memory allocations. Compilation times are increased by 10-15% when enabling the supercompiler.

The supercompiled results on these particular benchmarks are identical to the results reported in previous work for call-by-name languages by Wadler [57] and Sørensen et al. [47]. We do not provide any execution-time comparisons with these, though, since for identical intermediate representations after supercompilation, such measurements would only illustrate differences caused by back-end implementation techniques.

The work on Supero by Mitchell and Runciman [30] shows that there remain open problems when supercompiling large Haskell programs. These problems are mainly related to speed, both of the compiler and of the transformed program. When profiling Supero, Mitchell and Runciman found that a majority of the time was spent on their homeomorphic embedding test. Our transformation performs the corresponding test on a smaller part of the abstract syntax tree, so there is reason to believe that this will result in less time spent on testing homeomorphic embedding even on large programs for our transformation. The complexity of the homeomorphic embedding relation has been investigated by Narendran and Stillman [31], and they give an algorithm of complexity $O(size(e) \times size(f))$ for deciding whether $e \trianglelefteq f$. We expect essentially 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 transformation.

Benchmark	Time		Binary size	
	Before	After	Before	After
Double Append	105,844,704	89,820,912	89,484	90,800
Factorial	21,552	21,024	88,968	88,968
Flip a Tree	2,131,188	237,168	95,452	104,704
Sum of Squares of a Tree	276,102,012	28,737,648	95,452	104,912
Kort's Raytracer	12,050,880	7,969,224	91,968	91,460

Table 1: Time and size measurements

Benchmark	Allocations		Alloc Size	
	Before	After	Before	After
Double Append	270,035	180,032	2,160,280	1,440,256
Factorial	9	9	68	68
Flip a Tree	20,504	57	180,480	620
Sum of Squares of a Tree	4,194,338	91	29,360,496	908
Kort's Raytracer	60,021	17	320,144	124

Table 2: Allocation measurements

Benchmark	Not Supercompiled		Supercompiled	
	-c	–make	-c -S	–make -S
Double Append	0.183	0.300	0.202	0.319
Factorial	0.095	0.213	0.097	0.216
Flip a Tree	0.211	0.223	0.230	0.347
Sum of Squares of a Tree	0.214	0.332	0.234	0.349
Kort's Raytracer	0.239	0.359	0.278	0.399

Table 3: Compilation times

6.1. **Double Append.** As previously seen, supercompiling the appending of three lists saves one traversal over the first list. This is an example by Wadler [57], and the intermediate structure is fused away by our supercompiler. The program is:

$append\ xs\ ys\ =$ **case** xs **of**
$$[]\ \rightarrow\ ys$$
$$(x' : xs')\ \rightarrow\ x' : (append\ xs'\ ys)$$

$main\ xs\ ys\ zs =\ append\ (append\ xs\ ys)\ zs$

Supercompiling this program gives the same result that we obtained manually in Section 2:

$h_1\ xs_1\ ys_1\ zs_1\ =$ **case** xs_1 **of**
$$[]\ \rightarrow\ \textbf{case}\ ys_1\ \textbf{of}$$
$$[]\ \rightarrow\ zs_1$$
$$(y_1' : ys_1')\ \rightarrow\ y_1' : (h_2\ ys_1'\ zs_1)$$
$$(x_1' : xs_1')\ \rightarrow\ x_1' : (h_1\ xs_1'\ ys_1\ zs_1)$$
$h_2\ xs_2\ ys_2\ \ \ \ \ =$ **case** xs_2 **of**
$$[]\ \rightarrow\ ys_2$$
$$(x_2' : xs_2')\ \rightarrow\ x_2' : (h_2\ xs_2'\ ys_2)$$

$main\ xs\ ys\ zs =h_1\ xs\ ys\ zs$

In this measurement, three strings of 9000 characters each were appended to each other into a 27 000 character string. As can be seen in Table 2, the number of allocations goes down as one iteration over the first string is avoided. The binary size increases 1316 bytes, on a binary of roughly 90k.

6.2. **Factorial.** There are no intermediate lists created in a standard implementation of a factorial function, so any performance improvements must come from inlining or static reductions.

$fac\ 0\ = 1$
$fac\ n\ = n * fac\ (n - 1)$

$main\ = show\ (fac\ 3)$

The program is transformed to:

$h\ 0\quad = 1$
$h\ n\quad = n\ *\ h\ (n\ -\ 1)$

$main\ = show\ (3 * h\ 2)$

One recursion and a couple of reductions are eliminated, thereby slightly reducing the run-time. 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 [57], and just like Wadler we perform a double flip (thus restoring the original tree) before printing the total sum of all leaves.

$data\ Tree\ a\ =\ Leaf\ a\ |\ Branch\ (Tree\ a)\ (Tree\ a)$

$sumtr\ (Leaf\ a)\ =\ a$
$sumtr\ (Branch\ l\ r)\ =\ sumtr\ l\ +\ sumtr\ r$

$flip\ (Leaf\ x)\ =\ Leaf\ x$
$flip\ (Branch\ l\ r)\ =\ Branch\ (flip\ r)\ (flip\ l)$

$main\ xs\ =\ let\ ys\ =\ (flip\ (flip\ xs))\ in\ show\ (sumtr\ ys)$

This is transformed into:

$h\ t\ =$ **case** t **of**
$\qquad Leaf\ d\ \to\ d$
$\qquad Branch\ l\ r\ \to\ (h\ l)\ +\ (h\ r)$
$main\ xs\ =\ show\ ($ **case** xs **of**
$\qquad\qquad\qquad Leaf\ d\ \to\ d$
$\qquad\qquad\qquad Branch\ l\ r\ \to\ (h\ l)\ +\ (h\ r)\ )$

A binary tree of depth 12 was used in the measurement. The function h is isomorphic to $sumtr$ in the input program, and the double flip has been eliminated. Both the total number of allocations and the total size of allocations is reduced. The run-time is reduced by an order of magnitude. The binary size increases by about 10%, though.

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 [57].

data Tree a = *Leaf a* | *Branch* (*Tree a*) (*Tree a*)

square :: *Int* → *Int*
square x = *x* ∗ *x*

sumtr (*Leaf x*) = *x*
sumtr (*Branch l r*) = *sumtr l* + *sumtr r*

squaretr (*Leaf x*) = *Leaf* (*square x*)
squaretr (*Branch l r*) = *Branch* (*squaretr l*) (*squaretr r*)

main xs = *show* (*sumtr* (*squaretr xs*))

This is transformed to:

h t = **case** *t* **of**
       Leaf d → *d* ∗ *d*
       Branch l r → (*h l*) + (*h r*)

main xs = *show* (**case** *xs* **of**
             Leaf d → *d* ∗ *d*
             Branch l r → (*h l*) + (*h r*)

Almost all allocations are removed by our supercompiler, but the binary size is increased by nearly 10%. The run-time is improved by an order of magnitude.

6.5. **Kort's Raytracer.** The inner loop of a raytracer [22] written in Haskell is extracted and transformed.

zipWith f (*x* : *xs*) (*y* : *ys*) = (*f x y*) : *zipWith f xs ys*
zipWith ___ = []

sum :: [*Int*] → *Int*
sum [] = 0
sum (*x* : *xs*) = *x* + *sum xs*

main xs ys = *sum* (*zipWith* (∗) *xs ys*)

The transformed result is:

h xs ys = **case** *xs* **of**
       (*x'* : *xs'*) → **case** *ys* **of**
               (*y'* : *ys'*) → (*x'* ∗ *y'*) + (*h xs' ys'*)
               _ → 0
       _ → 0
main xs ys = *h xs ys*

The total run-time, the number of allocations, the total size of allocations and the binary size all decrease.

7. Related Work

There is much literature concerning algorithms that remove intermediate structures in functional programs. However, most of these works are in the the context of call-by-name or call-by-need languages, which makes the task of supercompilation 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 a call-by-name or call-by-need perspective.

7.1. **Lightweight Fusion.**

Ohori's and Sasano's Lightweight Fusion [35] works by promoting functions through the fix-point operator and guarantees termination by limiting inlining to at most once per function. They implement their transformation in a compiler for a variant of 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 themselves are not very similar. 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 or mutually recursive functions, something the positive supercompiler handles gracefully.

Despite the early stage of their work, Ohori and Sasano are proposing an interesting approach that appears quite powerful.

7.2. **Deforestation.**

Deforestation was pioneered by Wadler [57] for a first order language more than fifteen years ago. The function macros supported by the initial deforestation algorithm were not capable of fully emulating higher-order functions.

Marlow and Wadler [27] addressed the first-order restriction in a subsequent article [27]. This work was refined in Marlow's [1995] dissertation, where he also related deforestation to the cut-elimination principle of logic. Chin [5] 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 [14] and Marlow [28] have proven that their deforestation algorithms terminate. More recent work by Hamilton [15] extends deforestation with a treeless form that is easy to recognise and handles a wide range of functions, giving more transparency for the programmer.

Alimarine and Smetsers [2] have improved the producer and consumer analyses in Chin's [1994] algorithm to be based on semantics rather than syntax. They show that their algorithm can remove much of the overhead introduced by generic programming [16].

While these works are algorithmically rather close to ours due to the close relationship between deforestation and positive supercompilation, it supposes either a call-by-name or call-by-need context, and is thus not applicable to the kind of languages we target.

7.3. **Supercompilation.**

Closely related to deforestation is *supercompilation* [52, 53, 54, 55]. Supercompilation both removes intermediate structures and 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 [56]. The supercompiler Scp4 [33] is implemented in Refal and is the most well-known implementation from this line of work.

The *positive supercompiler* [47] is a variant which only propagates positive information such as inferred equalities between terms. The propagation is done by unification and the work highlights how similar deforestation and positive supercompilation really are. Narrowing-driven partial evaluation [3, 1] is the functional logic programming equivalent of positive supercompilation but formulated as a term rewriting system. Their approach also deals with non-determinism from backtracking, which makes the corresponding algorithms more complicated.

Strengthening the information propagation mechanism to propagate not only positive, but also negative information, yields *perfect supercompilation* [42, 43]. Negative information is the opposite of positive information, namely inequalities. These inequalities can be used to prune case-expression branches known not to be applicable, for example.

More recently, Mitchell and Runciman [30] have worked on supercompiling Haskell. They report run-time reductions of up to 55% when their supercompiler is used in conjunction with GHC.

Supercompilation has seen applications beyond program optimization: verification of cache coherence protocols [26] and proving term equivalence [21] are two examples. We do not believe that our supercompiler is useful for these applications since it is inherently weaker than the corresponding supercompiler with call-by-name semantics.

The positive supercompiler by Sørensen et al. [47] is the immediate ancestor of our work, although we have extended it to a higher-order language and converted it to work correctly for call-by-value languages.

7.4. **Generalized Partial Computation.** GPC [9, 50] 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.

The theorem prover is applied whenever a test is encountered, in order to determine which subset of the execution branches can actually be taken. Information about the predicate that 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. [10] have applied GPC in a call-by-value setting in a system called WS-DFU (Waseda Simplify-Distribute-Fold-Unfold), and report many successful experiments where optimal or near optimal residual programs are produced. It is unclear whether WS-DFU preserves termination behavior 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 [50] are very similar to the positive supercompiler, but the requirement for a theorem prover 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 similarities between GPC and positive supercompilation it should be straightforward to convert GPC to a call-by-value setting.

7.5. **Other Transformations.** Considering the vast amount of research conducted on program transformations in general, we only briefly survey other related transformations.

7.5.1. *Partial Evaluation.* Partial evaluation [18] is another instance of Burstall and Darlington's [1977] informal class of fold/unfold transformations.

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

Partial evaluation does not remove intermediate structures, something we deem necessary to enable the programmer to write programs in the clear and concise listful style. Both deforestation and supercompilation simulate call-by-name evaluation in the transformer, whereas partial evaluation simulates call-by-value. It is suggested by Sørensen et al. [46] that this might affect the strength of the transformation.

7.5.2. *Short Cut Fusion.* Short cut deforestation [12, 13] takes a different approach to deforestation, sacrificing some generality by only working on lists.

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

$$foldr\ f\ c\ (build\ g)\ =\ g\ f\ c$$

It is the responsibility of 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 [37]. Around a dozen benchmarks improved by more than 5%, the average was 3% and only one example got noticeably worse, by 1%. Heap allocations were reduced, by 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 currently contains a variant of the short cut deforestation implemented using rewrite rules [38].

Takano and Meijer [51] generalized short cut deforestation to work for any algebraic datatype through the acid rain theorem. Ghani and Johann [11] have also generalized the *foldr/build* rule to a *fold/superbuild* rule that can eliminate intermediate structures of inductive types without disturbing the contexts in which they are situated.

Launchbury and Sheard [24] worked on automatically transforming programs into suitable form for shortcut deforestation. Onoue et al. [36] showed an implementation of the acid rain theorem for Gofer where they could automatically transform recursive functions into a form suitable for shortcut fusion.

Chitil [6] used type-inference to transform the producer of lists into the abstracted form required by short cut deforestation. 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 also 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 of the list abstraction algorithm is that a *foldr* consumer can be fused with nearly any producer. One 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 by Chitil ensures that these list constructors are moved to the producer in order to make list abstraction possible.

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

Takano and Meijer [51] noted that the *foldr/build* rule for short cut deforestation has a dual. This is the *destroy/unfoldr* rule used in Zip Fusion [48], 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 removes all intermediate lists in *zip* $[1..n]$ $[1..n]$, addressing one of the main criticisms against the *foldr/build* rule. The technique can also remove intermediate lists from functions which consume their lists using accumulating parameters, which is usually a problematic case. The *destroy/unfoldr* rule is defined as:

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

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

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

8. Conclusions

We have presented a positive supercompiler for a higher-order call-by-value language and proven it correct with respect to call-by-value semantics. The adjustments required to preserve the termination properties of call-by-value evaluation are new and work well for many examples in the literature intended to show the usefulness of call-by-name transformations.

8.1. **Future Work.** We believe that the linearity restriction of rule R14 is not necessary for the soundness of our transformation, but have not yet found a way to prove this. This is a natural topic for future work, as is an investigation of whether the concept of an *inlining budget* may be used to control the balance between supercompilation benefits and code size.

More work could be done on the 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.

The supercompiler described in this article can be said to supersede several of the standard transformations commonly implemented by optimizing compilers, such as copy propagation, constant folding and basic inlining. We conjecture that this range could be extended to include transformations like common subexpression elimination as well, by

means of moderately small algorithm changes. An investigation of the scope for such generalizations is an important area of future research.

Acknowledgements

References

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

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

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

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

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

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

[7] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2.

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

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

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

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

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

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

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

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

[16] R. Hinze. *Generic Programs and Proofs*. Habilitationsschrift, Bonn University, 2000.

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

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

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

[20] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009.

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

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

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

[24] J. Launchbury and T. Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *FPCA*, pages 314–323, 1995.

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

[26] A. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

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

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

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

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

[31] P. Narendran and J. Stillman. On the Complexity of Homeomorphic Embeddings. Technical Report 87-8, Computer Science Department, State Univeristy of New York

at Albany, March 1987.

[32] C. St. J. A. Nash-Williams. On well-quasi-ordering finite trees. *Proceedings of the Cambridge Philosophical Society*, 59(4):833–835, October 1963.

[33] A. P. Nemytykh. The supercompiler SCP4: General structure. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, volume 2890 of *LNCS*, pages 162–170. Springer, 2003. ISBN 3-540-20813-5.

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

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

[36] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In R. S. Bird and L. G. L. T. Meertens, editors, *Algorithmic Languages and Calculi, IFIP TC2 WG2.1 International Workshop on Algorithmic Languages and Calculi, 17-22 February 1997, Alsace, France*, volume 95 of *IFIP Conference Proceedings*, pages 76–106. Chapman & Hall, 1997. ISBN 0-412-82050-1.

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

[38] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW'2001), 2nd September 2001, Firenze, Italy.*, Electronic Notes in Theoretical Computer Science, Vol 59. Utrecht University, September 28 2001. UU-CS-2001-23.

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

[40] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

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

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

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

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

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

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

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

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

[49] D. Syme. The F# programming language, June 2008. URL http://research.microsoft.com/fsharp.

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

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

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

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

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

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

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

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

Appendix A. Proofs

We borrow a couple of technical lemmas from Sands [39], and adapt the proofs to be valid under call-by-value:

Lemma A.1 (Sands, p. 24)**.** *For all expressions e and value substitutions θ such that $h \notin dom(\theta)$, if $e_0 \mapsto^1 e_1$ then*

$$\textbf{letrec } h = \lambda\overline{x}.e_1 \textbf{ in } [\theta(e_0)/z]e \trianglelefteq\trianglerighteq \textbf{letrec } h = \lambda\overline{x}.e_1 \textbf{ in } [h\,\theta(\overline{x})/z]e$$

Proof. Expanding both sides according to the definition of letrec yields:

$$(\lambda h.[\theta(e_0)/z]e)\,(\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n) \trianglelefteq\trianglerighteq (\lambda h.[h\,\theta(\overline{x})/z]e)\,(\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n)$$

and evaluating both sides one step $\mapsto$ gives:

$$[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n/h][\theta(e_0)/z]e \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n/h][h\,\theta(\overline{x})/z]e$$

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

$$[\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n/h]\theta e_0 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n/h]h\,\theta(\overline{x})$$

The substitution θ can safely be moved out since $h \notin dom(\theta)$:

$$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_0 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta(h\,\overline{x})$$

Performing evaluation steps on both sides yield:

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_0 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta(h\,\overline{x})$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_0 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta((\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n)\,\overline{x})$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_0 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta(fix\,(\lambda h.\lambda \overline{x}.e_1)\,\overline{x})$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta((\lambda f.f\,(\lambda n.fix\,f\,n))\,(\lambda h.\lambda \overline{x}.e_1)\,\overline{x})$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta((\lambda h.\lambda \overline{x}.e_1)\,(\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n)\,\overline{x})$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta((\lambda \overline{x}.e_1)\,\overline{x})$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1 \trianglelefteq\trianglerighteq [\overline{x}/\overline{x}][\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1$

$[\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1 \trianglelefteq\trianglerighteq [\lambda n.fix\,(\lambda h.\lambda \overline{x}.e_1)\,n/h]\theta e_1$

The LHS and the RHS are cost equivalent, so by Lemma 5.21 the initial expressions are cost equivalent. $\square$

Lemma A.2 (Sands, p. 25). $\rho'(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}) \trianglelefteq\trianglerighteq \mathbf{letrec}\,h = \lambda \overline{x}.\mathcal{R}\langle v\rangle\,\mathbf{in}\,\rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'})$

Proof (Similar to Sands [39]). By inspection of the rules for $\mathcal{D}[\![\,]\!]$, all free occurrences of h in $\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}$ must occur in sub-expressions of the form $h\,\overline{x}$. Suppose there are k such occurrences, which we can write as $\theta_1 h\,\overline{x}\ldots\theta_k h\,\overline{x}$, where the θ_i are just renamings of the variables $\overline{x}$. So $\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}$ can be written as $[\theta_1 h\,\overline{x}\ldots\theta_k h\,\overline{x}/z_1\ldots z_k]e'$, where e' contains no free occurrences of h. Then (substitution associates to the right):

$$
\begin{aligned}
\rho'(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}) \quad &\equiv \quad [\lambda \overline{x}.\mathcal{R}\langle g\rangle/h]\rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}) \\
&\trianglelefteq\trianglerighteq \quad [\lambda \overline{x}.\mathcal{R}\langle g\rangle/h]\rho([\theta_1 h\,\overline{x}\ldots\theta_k h\,\overline{x}/z_1\ldots z_k]e') \\
&\trianglelefteq\trianglerighteq \quad \rho([\theta_1 \mathcal{R}\langle g\rangle\ldots\theta_k \mathcal{R}\langle g\rangle/z_1\ldots z_k]e') \\
&\trianglelefteq\trianglerighteq \quad \text{(by Lemma A.1)} \\
&\qquad \mathbf{letrec}\,h = \lambda \overline{x}.\mathcal{R}\langle v\,\overline{e}\rangle\,\mathbf{in}\,\rho([\theta_1 h\,\overline{x}\ldots\theta_k h\,\overline{x}/z_1\ldots z_k]e') \\
&\equiv \quad \mathbf{letrec}\,h = \lambda \overline{x}.\mathcal{R}\langle v\rangle\,\mathbf{in}\,\rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\square,G,\rho'}) \qquad\square
\end{aligned}
$$

Lemma A.3. $\mathcal{R}\langle\mathbf{let}\,x = e\,\mathbf{in}\,f\rangle \trianglerighteq_s \mathbf{let}\,x = e\,\mathbf{in}\,\mathcal{R}\langle f\rangle$

Proof. Notice that $\mathcal{R}\langle\mathbf{let}\,x = \square\,\mathbf{in}\,f\rangle$ is a redex, and assume $e \mapsto^k v$. The LHS evaluates in k steps $\mathcal{R}\langle\mathbf{let}\,x = e\,\mathbf{in}\,f\rangle \mapsto^k \mathcal{R}\langle\mathbf{let}\,x = v\,\mathbf{in}\,f\rangle \mapsto \mathcal{R}\langle[v/x]f\rangle$, and the RHS evaluates in k steps $\mathbf{let}\,x = e\,\mathbf{in}\,\mathcal{R}\langle f\rangle \mapsto^k \mathbf{let}\,x = v\,\mathbf{in}\,\mathcal{R}\langle f\rangle \mapsto [v/x]\mathcal{R}\langle f\rangle$. Since contexts do not bind variables these two terms are equivalent and by Lemma 5.21 the initial terms are cost equivalent. $\square$

Lemma A.4. $\mathcal{R}\langle\mathbf{letrec}\,g = v\,\mathbf{in}\,e\rangle \trianglerighteq_S \mathbf{letrec}\,g = v\,\mathbf{in}\,\mathcal{R}\langle e\rangle$

Proof. Translate both sides by the definition of letrec into $\mathcal{R}\langle(\lambda g.e)\,(\lambda n.fix\,(\lambda g.v)\,n)\rangle \trianglerighteq_S (\lambda g.\mathcal{R}\langle e\rangle)\,(\lambda n.fix\,(\lambda g.v)\,n)$. Notice that $\mathcal{R}\langle\square\rangle$ is a redex. The LHS evaluates in 0 steps to $\mathcal{R}\langle(\lambda g.e)\,(\lambda n.fix\,(\lambda g.v)\,n)\rangle \mapsto \mathcal{R}\langle[\lambda n.fix\,(\lambda g.v)\,n/g]e\rangle$ and the RHS evaluates in 0 steps to $(\lambda g.\mathcal{R}\langle e\rangle)\,(\lambda n.fix\,(\lambda g.v)\,n) \mapsto [\lambda n.fix\,(\lambda g.v)\,n/g]\mathcal{R}\langle e\rangle$. Since our contexts do not bind variables these two terms are equivalent and by Lemma 5.21 the initial terms are cost equivalent. $\square$

Lemma A.5. $\mathcal{R}\langle\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}\rangle \trianglerighteq_s \mathbf{case}\,e\,\mathbf{of}\,\{p_i \to \mathcal{R}\langle e_i\rangle\}$

Proof. Notice that $\mathcal{R}\langle\mathbf{case}\,\square\,\mathbf{of}\,\{p_i \to e_i\}\rangle$ is a redex, and assume $e \mapsto^k n_j$. The LHS evaluates in k steps $\mathcal{R}\langle\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}\rangle \mapsto^k \mathcal{R}\langle\mathbf{case}\,n_j\,\mathbf{of}\,\{p_i \to e_i\}\rangle \mapsto \mathcal{R}\langle e_j\rangle$, and the RHS evaluates in k steps $\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to \mathcal{R}\langle e_i\rangle\} \mapsto^k \mathbf{case}\,n_j\,\mathbf{of}\,\{p_i \to \mathcal{R}\langle e_i\rangle\}\mathcal{R}\langle f\rangle \mapsto \mathcal{R}\langle e_j\rangle$. Since these two terms are equivalent the initial terms are cost equivalent by Lemma 5.21. $\square$

We set out to prove the main theorem about total correctness:

Theorem A.6 (Total Correctness). *Let $\mathcal{R}\langle e\rangle$ be an expression, and ρ an environment such that*

- *the range of ρ contains only closed expressions, and*
- *$fv(\mathcal{R}\langle e\rangle) \cap dom(\rho) = \emptyset$, and*

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

We reason by induction on the structure of expressions, and since the algorithm is total (Lemma 5.12) this coincides with inspection of each rule.

A.1. **R1.** We have that $\rho(\mathcal{D}[\![n]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{R}\langle n\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle n\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle n\rangle) = \mathcal{R}\langle n\rangle$ This is syntactically equivalent to the input, and we conclude $\mathcal{R}\langle n\rangle \trianglerighteq_s \rho(\mathcal{D}[\![n]\!]_{\mathcal{R},G,\rho})$.

A.2. **R2.** We have that $\rho(\mathcal{D}[\![x]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{R}\langle x\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle x\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle x\rangle) = \mathcal{R}\langle x\rangle$ This is syntactically equivalent to the input, and we conclude $\mathcal{R}\langle x\rangle \trianglerighteq_s \rho(\mathcal{D}[\![x]\!]_{\mathcal{R},G,\rho})$.

A.3. **R3.**

A.3.1. *Case: (1).*
Suppose $\exists h.\rho(h) \equiv \lambda\overline{x}.\mathcal{R}\langle g\rangle$ and hence that $\mathcal{D}[\![\mathcal{R}\langle g\rangle]\!]_{\square,G,\rho} = h\,\overline{x}$.

The conditions of the proposition ensure that $\overline{x} \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{D}[\![\mathcal{R}\langle g\rangle]\!]_{\square,G,\rho}) = \rho(h\,\overline{x}) = (\lambda\overline{x}.\mathcal{R}\langle g\rangle)\,\overline{x}$. However, $\mathcal{R}\langle g\rangle$ and $(\lambda\overline{x}.\mathcal{R}\langle g\rangle)\,\overline{x}$ are cost equivalent, which implies strong improvement, and we conclude $\mathcal{R}\langle g\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle g\rangle]\!]_{\square,G,\rho})$

A.3.2. *Case: (2).*
Suppose $\exists(h,t) \in \rho.t \trianglelefteq \mathcal{R}\langle g\rangle$ and that $\mathcal{R}\langle g\rangle \trianglelefteq t$, hence $\mathcal{D}[\![\mathcal{R}\langle g\rangle]\!]_{\square,G,\rho} = \mathcal{R}\langle g\rangle$.

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.3.3. *Case: (3).*

Suppose $\exists (h, t) \in \rho.t \trianglelefteq \mathcal{R}\langle g\rangle$ and hence that $\mathcal{D}[\![\mathcal{R}\langle g\rangle]\!]_{\Box, G, \rho} = [\mathcal{D}[\![\overline{f}]\!]_{\Box, G, \rho}/\overline{y}]\mathcal{D}[\![f_g]\!]_{\Box, G, \rho}$.

We have $\rho(\mathcal{D}[\![g]\!]_{\mathcal{R}, G, \rho}) = \rho([\mathcal{D}[\![\overline{f}]\!]_{\Box, G, \rho}/\overline{y}]\mathcal{D}[\![f_g]\!]_{\Box, G, \rho}) = [\rho(\mathcal{D}[\![\overline{f}]\!]_{\Box, G, \rho})/\overline{x}]\rho(\mathcal{D}[\![f_g]\!]_{\Box, G, \rho})$. By the induction hypothesis, $\overline{f} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{f}]\!]_{\Box, G, \rho})$ and $f_g \trianglerighteq_s \rho(\mathcal{D}[\![f_g]\!]_{\Box, G, \rho})$ and by congruence properties of strong improvement (Lemma 5.23:1) $\mathcal{R}\langle g\rangle \trianglerighteq_s \rho(\mathcal{D}[\![g]\!]_{\mathcal{R}, G, \rho})$.

A.3.4. *Case: (4a).* Analogous to the previous case.

A.3.5. *Case: (4b).*
If $\mathcal{D}[\![\mathcal{R}\langle g\rangle]\!]_{\Box, G, \rho} = \rho(\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'}\,\textbf{in}\, h\,\overline{x})$.
where $\rho' = \rho \cup (h, \lambda\overline{x}.\mathcal{R}\langle g\rangle)$ and $h \notin (\overline{x} \cup dom(\rho))$. We need to show that:

$$\mathcal{R}\langle g\rangle \trianglerighteq_s \rho(\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'}\,\textbf{in}\, h\,\overline{x})$$

Since $h, \overline{x} \notin dom(\rho)$ we have that $\rho(\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'}\,\textbf{in}\, h\,\overline{x}) \equiv \textbf{letrec}\, h = \lambda\overline{x}.\rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'})\,\textbf{in}\, h\,\overline{x}$.

$\mathcal{R}$ is a reduction context, hence $\mathcal{R}\langle g\rangle \mapsto^1 \mathcal{R}\langle v\rangle$. By Lemma A.1 we have that $\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\,\mathcal{R}\langle g\,\overline{e}\rangle \trianglelefteq\trianglerighteq \textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\, h\,\overline{x}$. Since $h \notin fv(\mathcal{R}\langle g\rangle)$ this simplifies to $\mathcal{R}\langle g\rangle \trianglelefteq\trianglerighteq \textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\, h\,\overline{x}$. It is necessary and sufficient to prove that

$$\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\, h\,\overline{x} \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.\rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'})\,\textbf{in}\, h\,\overline{x}$$

By Theorem 5.27 it is sufficient to show:

$$\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\,\mathcal{R}\langle v\rangle \trianglerighteq_s$$
$$\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\,\rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'})$$

By Lemma A.2 and $\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{R}\langle v\rangle\,\textbf{in}\,\mathcal{R}\langle v\rangle \trianglelefteq\trianglerighteq \mathcal{R}\langle v\rangle$, this is equivalent to showing that

$$\mathcal{R}\langle v\rangle \trianglerighteq_s \rho'(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho'})$$

Which follows from the induction hypothesis, since it is a shorter transformation.

A.3.6. *Case: (4c).* We have that $\rho(\mathcal{D}[\![g]\!]_{\mathcal{R}, G, \rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho})$. By the induction hypothesis $\mathcal{R}\langle v\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{\Box, G, \rho})$, and since $\mathcal{R}\langle g\rangle \mapsto^1 \mathcal{R}\langle v\rangle$ it follows from Lemma 5.23:4 that $\mathcal{R}\langle g\rangle \trianglerighteq_s \rho(\mathcal{D}[\![g]\!]_{\mathcal{R}, G, \rho})$.

A.4. **R4.** We have that $\rho(\mathcal{D}[\![k\,\overline{e}]\!]_{\Box, G, \rho}) = \rho(k\,\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho})$, and the conditions of the proposition ensure that $fv(k\,\overline{e}) \cap dom(\rho) = \emptyset$, so $\rho(k\,\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho}) = k\,\rho(\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho})$. By the induction hypothesis, $\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho})$, and by congruence properties of strong improvement (Lemma 5.23:1) $k\,\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![k\,\overline{e}]\!]_{\Box, G, \rho})$.

A.5. **R5.** We have that $\rho(\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R}, G, \rho}) = \rho(\mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho}\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle x\,\overline{e}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho}\rangle) = \mathcal{R}\langle x\,\rho(\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho})\rangle$. By the induction hypothesis, $\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{e}]\!]_{\Box, G, \rho})$, and by congruence properties of strong improvement (Lemma 5.23:1) $\mathcal{R}\langle x\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R}, G, \rho})$.

A.6. **R6.** We have that $\rho(\mathcal{D}[\![\lambda \overline{x}.e]\!]_{\square,G,\rho}) = \rho(\lambda \overline{x}.\mathcal{D}[\![e]\!]_{\square,G,\rho})$, and the conditions of the proposition ensure that $fv(\lambda \overline{x}.e) \cap dom(\rho) = \emptyset$, so $\rho(\lambda \overline{x}.\mathcal{D}[\![e]\!]_{\square,G,\rho}) = \lambda \overline{x}.\rho(\mathcal{D}[\![e]\!]_{\square,G,\rho})$. By the induction hypothesis, $e \unrhd_s \rho(\mathcal{D}[\![e]\!]_{\square,G,\rho})$, and by congruence properties of strong improvement (Lemma 5.23:1) $\lambda \overline{x}.e \unrhd_s \rho(\mathcal{D}[\![\lambda \overline{x}.e]\!]_{\square,G,\rho})$.

A.7. **R7.** We have that $\rho(\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{\square,G,\rho})$. By the induction hypothesis, $\mathcal{R}\langle n\rangle \unrhd_s \rho(\mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{\square,G,\rho})$, and since $\mathcal{R}\langle n_1 \oplus n_2\rangle \mapsto \mathcal{R}\langle n\rangle$ it follows from Lemma 5.23:4 that $\mathcal{R}\langle n_1 \oplus n_2\rangle \unrhd_s \rho(\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},G,\rho})$.

A.8. **R8.**

a) $e_1 \oplus e_2 = a$: We have that $\rho(\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![e_1]\!]_{\square,G,\rho} \oplus \mathcal{D}[\![e_2]\!]_{\square,G,\rho})$, by the given conditions $fv(\mathcal{R}\langle e_1 \oplus e_2\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle \mathcal{D}[\![e_1]\!]_{\square,G,\rho} \oplus \mathcal{D}[\![e_2]\!]_{\square,G,\rho}\rangle) = \mathcal{R}\langle \rho(\mathcal{D}[\![e_1]\!]_{\square,G,\rho}) \oplus \rho(\mathcal{D}[\![e_2]\!]_{\square,G,\rho})\rangle$. By the induction hypothesis $e_1 \unrhd_s \rho(\mathcal{D}[\![e_1]\!]_{\square,G,\rho})$ and $e_2 \unrhd_s \rho(\mathcal{D}[\![e_2]\!]_{\square,G,\rho})$, and by congruence properties of strong improvement (Lemma 5.23:1) $\mathcal{R}\langle e_1 \oplus e_2\rangle \unrhd_s \rho(\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},G,\rho})$.

b) $e_1 = n$ or $e_1 = a$: We have $\rho(\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![e_2]\!]_{\mathcal{R}\langle e_1 \oplus \square\rangle,G,\rho})$ and $\mathcal{R}\langle e_1 \oplus e_2\rangle \unrhd_s \rho(\mathcal{D}[\![e_2]\!]_{\mathcal{R}\langle e_1 \oplus \square\rangle,G,\rho})$ follows from the induction hypothesis.

c) otherwise: We have that $\rho(\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![e_1]\!]_{\mathcal{R}\langle \square \oplus e_2\rangle,G,\rho})$ and $\mathcal{R}\langle e_1 \oplus e_2\rangle \unrhd_s \rho(\mathcal{D}[\![e_1]\!]_{\mathcal{R}\langle \square \oplus e_2\rangle,G,\rho})$ follows from the induction hypothesis.

A.9. **R9.** We have that $\rho(\mathcal{D}[\![(\lambda \overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle \mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f\rangle]\!]_{\square,G,\rho})$. Evaluating the input term yields: $\mathcal{R}\langle(\lambda \overline{x}.f)\,\overline{e}\rangle \mapsto^r \mathcal{R}\langle(\lambda \overline{x}.f)\,\overline{v}\rangle \mapsto \mathcal{R}\langle[\overline{v}/\overline{x}]f\rangle$, and evaluating the input to the recursive call yields: $\mathcal{R}\langle \mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f\rangle \mapsto^r \mathcal{R}\langle \mathbf{let}\,\overline{x} = \overline{v}\,\mathbf{in}\,f\rangle \mapsto \mathcal{R}\langle[\overline{v}/\overline{x}]f\rangle$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 5.21 their ancestor terms are cost equivalent, $\mathcal{R}\langle(\lambda \overline{x}.f)\,\overline{e}\rangle \lhd\rhd \mathcal{R}\langle \mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f\rangle$, and cost equivalence implies strong improvement. By the induction hypothesis $\mathcal{R}\langle \mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f\rangle \unrhd_s \rho(\mathcal{D}[\![\mathcal{R}\langle \mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f\rangle]\!]_{\square,G,\rho})$, and therefore $\mathcal{R}\langle(\lambda \overline{x}.f)\,\overline{e}\rangle \unrhd_s \rho(\mathcal{D}[\![(\lambda \overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},G,\rho})$.

A.10. **R10.** We have $\rho(\mathcal{D}[\![e\,e']\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![e]\!]_{\mathcal{R}\langle \square\,e'\rangle,G,\rho})$ and $\mathcal{R}\langle e\,e'\rangle \unrhd_s \rho(\mathcal{D}[\![e]\!]_{\mathcal{R}\langle \square\,e'\rangle,G,\rho})$ follows from the induction hypothesis.

A.11. **R11.** We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x = n\,\mathbf{in}\,f]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle[n/x]f\rangle]\!]_{\square,G,\rho})$. By the induction hypothesis $\mathcal{R}\langle[n/x]f\rangle \unrhd_s \rho(\mathcal{D}[\![\mathcal{R}\langle[n/x]f\rangle]\!]_{\square,G,\rho})$, and since $\mathcal{R}\langle \mathbf{let}\,x = n\,\mathbf{in}\,f\rangle \mapsto \mathcal{R}\langle[n/x]f\rangle$ it follows from Lemma 5.23:4 that $\mathcal{R}\langle \mathbf{let}\,x = n\,\mathbf{in}\,f\rangle \unrhd_s \rho(\mathcal{D}[\![\mathbf{let}\,x = n\,\mathbf{in}\,f]\!]_{\mathcal{R},G,\rho})$.

A.12. **R12.** We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x = y\,\mathbf{in}\,f]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{\square,G,\rho})$. By the induction hypothesis $\mathcal{R}\langle[y/x]f\rangle \unrhd_s \rho(\mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{\square,G,\rho})$, and since $\mathcal{R}\langle \mathbf{let}\,x = y\,\mathbf{in}\,f\rangle \lhd\rhd \mathcal{R}\langle[y/x]f\rangle$ it follows that $\mathcal{R}\langle \mathbf{let}\,x = y\,\mathbf{in}\,f\rangle \unrhd_s \rho(\mathcal{D}[\![\mathbf{let}\,x = y\,\mathbf{in}\,f]\!]_{\mathcal{R},G,\rho})$.

A.13. **R13.**

A.13.1. *Case:* $x \in strict(f)$. We have $\rho(\mathcal{D}[\![\mathbf{let}\, x = e \,\mathbf{in}\, f]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{\Box,G,\rho})$. Evaluating the input term yields $\mathcal{R}\langle\mathbf{let}\, x = e \,\mathbf{in}\, f\rangle \mapsto^r \mathcal{R}\langle\mathbf{let}\, x = v \,\mathbf{in}\, f\rangle \mapsto \mathcal{R}\langle[v/x]f\rangle \mapsto^s \mathcal{E}\langle v\rangle$, and evaluating the input to the recursive call yields: $\mathcal{R}\langle[e/x]f\rangle \mapsto^s \mathcal{E}\langle e\rangle \mapsto^r \mathcal{E}\langle v\rangle$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 5.21 their ancestor terms are cost equivalent, $\mathcal{R}\langle\mathbf{let}\, x = e \,\mathbf{in}\, f\rangle \trianglelefteq\trianglerighteq \mathcal{R}\langle[e/x]f\rangle$, and cost equivalence implies strong improvement. By the induction hypothesis $\mathcal{R}\langle[e/x]f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{\Box,G,\rho})$, and therefore $\mathcal{R}\langle\mathbf{let}\, x = e \,\mathbf{in}\, f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\, x = e \,\mathbf{in}\, f]\!]_{\mathcal{R},G,\rho})$.

A.13.2. *Case: otherwise.* We have that $\rho(\mathcal{D}[\![\mathbf{let}\, x = e \,\mathbf{in}\, f]\!]_{\mathcal{R},G,\rho}) = \rho(\mathbf{let}\, x = \mathcal{D}[\![e]\!]_{\Box,G,\rho} \,\mathbf{in}\, \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{\Box,G,\rho})$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle\mathbf{let}\, x = e \,\mathbf{in}\, f\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathbf{let}\, x = \mathcal{D}[\![e]\!]_{\Box,G,\rho} \,\mathbf{in}\, \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{\Box,G,\rho}) = \mathbf{let}\, x = \rho(\mathcal{D}[\![e]\!]_{\Box,G,\rho}) \,\mathbf{in}\, \rho(\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{\Box,G,\rho})$. By the induction hypothesis $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{\Box,G,\rho})$ and $\mathcal{R}\langle f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{\Box,G,\rho})$. By Lemma A.3 the input is strongly improved by $\mathbf{let}\, x = e \,\mathbf{in}\, \mathcal{R}\langle f\rangle$, and therefore $\mathcal{R}\langle\mathbf{let}\, x = e \,\mathbf{in}\, f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\, x = e \,\mathbf{in}\, f]\!]_{\mathcal{R},G,\rho})$.

A.14. **R14.** We have that $\rho(\mathcal{D}[\![\mathbf{letrec}\, g = v \,\mathbf{in}\, e]\!]_{\mathcal{R},G,\rho}) = \rho(\mathbf{letrec}\, g = v \,\mathbf{in}\, \mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{\Box,G,\rho})$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle\mathbf{letrec}\, g = v \,\mathbf{in}\, e\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathbf{letrec}\, g = v \,\mathbf{in}\, \mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{\Box,G,\rho}) = \mathbf{letrec}\, g = v \,\mathbf{in}\, \rho(\mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{\Box,G,\rho})$. By the induction hypothesis $\mathcal{R}\langle e\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{\Box,G,\rho})$. By Lemma A.4 the input is strongly improved by $\mathbf{letrec}\, g = v \,\mathbf{in}\, \mathcal{R}\langle e\rangle$, and therefore $\mathcal{R}\langle\mathbf{letrec}\, g = v \,\mathbf{in}\, e\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{letrec}\, g = v \,\mathbf{in}\, e]\!]_{\mathcal{R},G,\rho})$.

A.15. **R15.** We have $\rho(\mathcal{D}[\![\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho}) = \rho(\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho}\})$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to e_i\}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho}\}) = \mathbf{case}\, x \,\mathbf{of}\, \{p_i \to \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho})\}$. By the induction hypothesis $\mathcal{R}\langle e_i\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho})$. Using Lemma A.5 the input is strongly improved by $\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to \mathcal{R}\langle e_i\rangle\}$, and therefore $\mathcal{R}\langle\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, x \,\mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho})$.

A.16. **R16.** We have $\rho(\mathcal{D}[\![\mathbf{case}\, k_j \,\overline{e} \,\mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e} \,\mathbf{in}\, e_j\rangle]\!]_{\Box,G,\rho})$. Evaluating the input term yields $\mathcal{R}\langle\mathbf{case}\, k_j \,\overline{e} \,\mathbf{of}\, \{p_i \to e_i\}\rangle \mapsto^r \mathcal{R}\langle\mathbf{case}\, k_j \,\overline{v} \,\mathbf{of}\, \{p_i \to e_i\}\rangle \mapsto \mathcal{R}\langle[\overline{v}/\overline{x}_j]e_j\rangle$, and evaluating the input to the recursive call yields $\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e} \,\mathbf{in}\, e_j\rangle \mapsto^r \mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{v} \,\mathbf{in}\, e_j\rangle \mapsto \mathcal{R}\langle[\overline{v}/\overline{x}_j]e_j\rangle$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 5.21 their ancestor terms are cost equivalent, $\mathcal{R}\langle\mathbf{case}\, k_j \,\overline{e} \,\mathbf{of}\, \{p_i \to e_i\}\rangle \trianglelefteq\trianglerighteq \mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e} \,\mathbf{in}\, e_j\rangle$, and cost equivalence implies strong improvement. According to the induction hypothesis $\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e} \,\mathbf{in}\, e_j\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e} \,\mathbf{in}\, e_j\rangle]\!]_{\Box,G,\rho})$, and therefore $\mathcal{R}\langle\mathbf{case}\, k_j \,\overline{e} \,\mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, k_j \,\overline{e} \,\mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho})$.

A.17. **R17.** We have that $\rho(\mathcal{D}[\![\mathbf{case}\, n_j \,\mathbf{of}\, \{p_i \to e_i\}]\!]_{),G,\rho} = \rho(\mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{\Box,G,\rho})$. By the induction hypothesis $\mathcal{R}\langle e_j\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{\Box,G,\rho})$, and since $\mathcal{R}\langle\mathbf{case}\, n_j \,\mathbf{of}\, \{p_i \to e_i\}\rangle \mapsto \mathcal{R}\langle e_j\rangle$ it follows from Lemma 5.23:4 that $\mathcal{R}\langle\mathbf{case}\, n_j \,\mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, n_j \,\mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho})$.

A.18. **R18.** We have that $\rho(\mathcal{D}[\![\mathbf{case}\,a\,\mathbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho}) = \rho(\mathbf{case}\,\mathcal{D}[\![a]\!]_{\Box,G,\rho}\,\mathbf{of}\,\{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho}\})$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle\mathbf{case}\,a\,\mathbf{of}\,\{p_i \to e_i\}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathbf{case}\,\mathcal{D}[\![a]\!]_{\Box,G,\rho}\,\mathbf{of}\,\{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho}\}) = \mathbf{case}\,\rho(\mathcal{D}[\![a]\!]_{\Box,G,\rho})\,\mathbf{of}\,\{p_i \to \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho})\}$. By the induction hypothesis $a \trianglerighteq_s \rho(\mathcal{D}[\![a]\!]_{\Box,G,\rho})$ and $\mathcal{R}\langle e_i\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{\Box,G,\rho})$ and by Lemma A.5 the input is strongly improved by $\mathbf{case}\,a\,\mathbf{of}\,\{p_i \to \mathcal{R}\langle e_i\rangle\}$, and therefore $\mathcal{R}\langle\mathbf{case}\,a\,\mathbf{of}\,\{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\,a\,\mathbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho})$.

A.19. **R19.** We have that $\rho(\mathcal{D}[\![\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{D}[\![e]\!]_{\mathcal{R}\langle\mathbf{case}\,\Box\,\mathbf{of}\,\{p_i\to e_i\}\rangle,G,\rho})$ and $\mathcal{R}\langle\mathbf{case}\,e\,\mathbf{of}\,\{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{\mathcal{R}\langle\mathbf{case}\,\Box\,\mathbf{of}\,\{p_i\to e_i\}\rangle,G,\rho})$ follows from the induction hypothesis.

A.20. **R20.** We have that $\rho(\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho}) = \rho(\mathcal{R}\langle e\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle e\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle e\rangle) = \mathcal{R}\langle e\rangle$ This is syntactically equivalent to the input, and we conclude $\mathcal{R}\langle e\rangle \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{\mathcal{R},G,\rho})$.