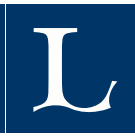
2008:22

LICENTIATE THESIS



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

Peter A Jonsson

Luleå University of Technology
Department of Computer Science and Electrical Engineering
EISLAB

2008:22 | ISSN: 1402-1757 | ISRN: LTU-LIC -- 08/22 -- SE

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

Peter A. Jonsson

EISLAB
Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

Supervisor:

Johan Nordlander

ABSTRACT

Intermediate structures such as lists and higher-order functions are very common in most styles of functional programming. While allowing the programmer to write clear and concise programs, the creation and destruction of these structures impose a run time overhead which is not negligible. Deforestation algorithms is a family of program transformations that remove these intermediate structures in an automated fashion, thereby improving program performance.

While there has been plenty of work on deforestation-like transformations that remove intermediate structures for languages with call-by-name semantics, no investigations have been performed for call-by-value languages. It has been suggested that existing call-by-name algorithms could be applied to call-by-value programs, possibly introducing termination in the program. This hides looping bugs from the programmer, and changes the behaviour of a program depending on whether it is optimized or not.

We present a transformation, positive supercompilation, for a higher-order call-by-value language that preserves termination properties of the programs it is applied to. We prove the algorithm correct and compare it to existing call-by-name transformations. Our results show that deforestation-like transformations are both possible and useful for call-by-value languages, with speedups up to an order of magnitude for certain benchmarks.

Our algorithm is particularly important in the context of embedded systems where resources are scarce. By both removing intermediate structures and performing program specialization the footprint of programs can shrink considerably without any manual intervention by the programmer.

CONTENTS

FIGURES

TABLES

PREFACE

First and foremost, I would like to thank my supervisor Johan Nordlander. Without his guidance and support, the work presented in this thesis would have been very different.

I would like to thank all my colleagues at the division of EISLAB, without you this research would not have been possible to conduct. Viktor Leijon, with whom I have shared an office for the past two years deserves a special mention – he has read many of my drafts and we have discussed numerous ideas over the years. Martin Kero has also been a great source of ideas and inspiration.

The administrative support at the Department of Computer Science and Electrical Engineering also deserves a special mention. All the help I have gotten through the years is invaluable, and without your help this work would have been much delayed, perhaps not even possible.

There are other contributors to this work as well, in particular I would like to thank Simon Marlow, Duncan Coutts and Neil Mitchell for valuable discussions regarding research on program transformations.

CHAPTER 1

Introduction

1.1 Our Thesis

Positive supercompilation is feasible and useful for a higher-order call-by-value language. Call-by-value is the most common evaluation strategy in programming languages today.

High levels of abstraction, the possibility to reason about components of software in isolation, and the ability to compose different software components together are crucial features to improve productivity in software development (Hughes 1989). A pure functional language (Sabry 1998) gives the ability of equational reasoning about programs, along with features such as higher-order functions that aid programmer productivity.

It is however difficult to implement higher-order languages efficiently. In 1991, the state of the art Scheme compilers produced code that was roughly comparable to what a simple non-optimising C compiler produced (Shivers 1991). The problem of competing performance-wise with the output of a C compiler is still existent today, in 2008. Progress has been made, but the functional languages are still playing catch-up with the compilers for more traditional imperative languages. The Great Language Shootout (Fulgham 2007) is a good example, a set of benchmarks over a set of compilers, where GCC almost always comes out on top.

Two of the identified problems are temporary intermediate structures (Wadler 1990), that are used to pass data between functions, and higher-order functions that make frequent calls to functions passed as parameters which is costly on a modern processor. While there has been plenty of research in a call-by-name context for these problems, nothing with call-by-value semantics has been published. Applying existing transformations to call-by-value languages has been suggested, possibly altering the semantics by introducing termination in the process.

This is no longer necessary, as this thesis develops an algorithm for removing many of these intermediate structures and higher-order functions at compile time in a call-by-value language. We demonstrate that a naïve conversion of previous algorithms to call-by-value semantics yields worse results than necessary. Our algorithm is proven correct and extensions are suggested for future work. The algorithm is implemented in an experimental compiler for Timber (Carlsson et al. 2003), a pure object-oriented functional language. Measurements show that certain programs can be sped up by an order of magnitude.

1.2 General Outline

Chapter 1: Introduction The thesis is introduced along with an explaination of the problem and a motivation for it being solved.

Chapter 2: State of Art A survey of the state of the art within the field of program transformations that perform specialization and intermediate data structure removal.

Chapter 3: Higher order Supercompilation An algorithm that specializes programs and automatically removes intermediate data structures for a higher-order call-by-value language is presented. It is proven correct, and important design issues for the algorithm is presented. This is the central core that supports our thesis.

Chapter 4: Measurements Measurements from a set of common examples from the litterature on program transformations. It is shown that our positive supercompiler does remove intermediate structures, and can improve the performance by an order of magnitude for certain benchmarks.

Chapter 5: Conclusions Conclusions are drawn from the work presented in previous chapters, a summary of contributions of this work and a discussion of further work.

1.3 Program Specialization

A function that takes several arguments can be specialized by freezing one or more arguments to a special value. In mathematical analysis this is called *projection* or *restriction*, in logic it is called *currying*. Consider the example of a function that computes x^y defined as:

$$\begin{aligned} pow\ x\ y = {} & \mathbf{if}\ y\ =\ 0\ \mathbf{then} \\ & \quad 1 \\ & \mathbf{else} \\ & \quad x\ *\ pow\ x\ (y\ -\ 1) \end{aligned}$$

If the exponent is statically known at compile time, a more efficient version of pow can be created. If the value is 3, the function pow_3 which takes one argument can be created:

$$pow_3\ x = x * x * x$$

This new function does not contain any recursion, and will therefore be cheaper to evaluate. The essence of partial evaluation (Jones et al. 1993) is specializing programs rather than functions. The output program, which is specialized with respect to the statically known data, is called a residual program.

In higher-order languages, functions are first class citizens – they can be passed as parameters to other functions, and returned as values from functions. Typical higher-order functions are map, $filter$ and $foldr$. For example, a function $lowers$ which uses higher-order functions from the prelude to return all the lower case characters in a list is defined as:

$$\mathit{filter}\ p\ xs = [\ x \mid x\ <-\ xs,\ p\ x\ ]$$

$$\mathit{lowers}\ xs\ = \mathit{filter}\ \mathit{isLower}\ xs$$

This could be specialized down to a first-order version:

$$\mathit{filter}_{\mathit{isLower}}\ xs = [\ x \mid x\ <-\ xs,\ \mathit{isLower}\ x\ ]$$

$$\mathit{lowers}\ xs\qquad = \mathit{filter}_{\mathit{isLower}}\ xs$$

which is less costly, since it does not contain any higher-order functions. The challenge for a program specializer is to evaluate as much of a program as possible at compile time, only leaving in computations that are not possible to perform at compile time.

Implementation-wise, closures are the mechanism that makes it possible for the compiler to compile code to pass around functions. Other implementation methods exist, but nothing else is as widely used as closures (Danvy and Nielsen 2001).

Closures reside on the heap, and consist of a code pointer to the function in question and the free variables of that function. Since the heap is garbage collected, the closures are a burden for the garbage collector. A copying garbage collector (Cheadle et al. 2004), which is common in functional languages, has to copy the closures back and forth during their life time.

To execute the function inside the closure, an indirect jump through the code pointer has to be performed. This is a costly operation on a modern superscalar processor with a deep pipeline. Specializing the program by removing higher-order functions gives several advantages:

- No indirect jumps necessary to evaluate the functions.
- No need to create and destruct closures, which puts less stress on the garbage collector.
- Functions can be specialized to take fewer arguments, which in turn might make all the parameters passed in registers instead of on the stack. This gives fewer memory reads and writes.

1.4 Intermediate Structures

A typical example of programming in the listful style is to compute the sum of the squares of the numbers from 1 to n by:

$$\mathit{sum}\ (\mathit{map}\ \mathit{square}\ [1..n])$$

Wadler (1990) eloquently described this style as "the use of functions to encapsulate common patterns of computation". The intermediate lists are the glue that hold these functions together. By transforming the program one could instead obtain:

$$h\ 0\ 1\ n$$

```
  where
    h a m n = if m > n then
                a
              else
                h (a + square m) (m + 1) n
```

which is more efficient because all operations on list cells have been eliminated, thereby avoiding both construction and garbage collection of any intermediate lists. In other words, a transformation of the program incurs savings in both time and space usage.

Much research has targeted this problem, and Burstall and Darlington's (1977) informal class of fold/unfold-transformations is one important early work in the field. The idea is that programs are altered by small local correctness-preserving transformations to become more efficient. The system they described was guided by a human, relying on the programmer to realize where it would be beneficial to do a transformation step.

The example above is from Wadler's work on deforestation (1990), an automatic algorithm that removes intermediate structures. Previous to this work, Wadler had been working on listlessness (Wadler 1984, 1985), restricting the language enough to make the list removal transformation easy to perform.

```
sumSq n      = sum (map square [1..n])
square x     = x * x

fastSumSq n = fastSumSq' 1
  where
    fastSumSq' x = if x > n then
                     0
                   else
                     square x + fastSumSq' (x + 1)
```

Figure 1.1: Sum of squares from 1 to n

While the listful style allows the programmer to write clear and concise programs, the creation and destruction of these intermediate structures impose a run time overhead which is not negligible. Gill (1996, p. 6) has measured the number of reduction steps (Table 1.1) under Hugs for the programs in Figure 1.1 and Figure 1.2. The number of reduction steps for the listful version of the program is greater than for the listless version. Although these measurements are for a lazy (call-by-need) language, they still give an indication that listful programming carries a cost compared to the listless programming.

If the compiler could automatically remove those intermediate structures it would allow the programmer to write programs in the most clear style, without concern for efficiency. The need for intermediate structure removal in a call-by-value language is in fact even greater than in a

$$
\begin{aligned}
&natural \quad = reverse \;.\; map\ (`mod`\ 10)\;.\; takeWhile\ (\neq\ 0)\;.\; iterate\ (`div`\ 10)\\
&\\
&fastNatural\ n = fastNatural'\ n\ []\\
&\quad \textbf{where}\\
&\quad\quad fastNatural'\ n\ m = \textbf{if}\ n\ \neq\ 0\ \textbf{then}\\
&\quad\quad\quad\quad\quad\quad\quad\quad fastNatural'\ (n\ `div`\ 10)\ (n\ `mod`\ 10 : m)\\
&\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad m
\end{aligned}
$$

Figure 1.2: Obtain a list of a number's of digits

Table 1.1: Reduction steps of listful and listless versions

Example	Parameter	Listful (reductions)	Listless (reductions)
Sum of squares	10	123	73
Digits of a Natural	1234	82	44

call-by-need language – complete intermediate structures need to stay on the heap since they are not produced and consumed lazily. In short, removing intermediate structures gives several benefits:

- No need to create and destruct the intermediate cells of those structures. This will save running time for the garbage collector since it has to perform less work in total.
- No memory accesses needed for those removed intermediate structures, neither for reading nor writing.
- The fusion of functions might open up for additional optimisations of the program due to the surrounding context.

1.5 Preserving Semantics

Some programs that terminate under call-by-name do not terminate under call-by-value. A typical example are functions that ignore their arguments, consider the function *const* called from *main*:

$$const\ x\ y\ =\ x$$

$$main\ =\ \ const\ 1\ \bot$$

Transforming this to the new program

$$main = 1$$

is not a semantically sound transformation under call-by-value! Under call-by-name this evaluates to 1. However, this result is not possible to reach under call-by-value semantics, since there is obvious non-termination in the original code snippet. The call-by-value semantics force worse performance; the best transformation result that still preserves semantics is $\mathbf{let}\ x = \bot\ \mathbf{in}\ 1$.

The inherent conflict between having call-by-value semantics and delaying evaluation of arguments as long as possible is the core of the problem we face when trying to construct an equally powerful call-by-value deforestation algorithm as the previous call-by-name ones.

The transformation we study, which will remove intermediate structures, perform program specialization and preserve call-by-value semantics, is positive supercompilation as outlined in Chapter 3. This brings us back to our thesis: positive supercompilation is feasible and useful for a higher-order call-by-value language. It will automatically remove many intermediate structures as well as specialize programs.

CHAPTER 2

State of the Art in Program Specialization

> To know the road ahead,
> ask those coming back.
> – *Chinese proverb*

This chapter will give an overview of the state of the art in program specialization. The comparison and critical analysis is postponed until Section 3.8, right after the technical contributions of this thesis have been presented in detail.

A first order call-by-name language is used to demonstrate different transformation algorithms in this chapter. The examples have quite a few small details left out, but should convey the essence each transformation. Sørensen et al. (1994) has made a very similar comparison, and for further details the reader is referred to their work. The grammar of the language we use is:

$$\begin{array}{rcll} e & ::= & x \mid g\,\overline{e} & \text{(Variable; Application)} \\ & \mid & k\,\overline{e} \mid \textbf{case}\,e\,\textbf{of}\,\{p_i \to e_i\} & \text{(Constructor; Case expression)} \\ \\ p & ::= & k\,\overline{x} & \end{array}$$

We let constructor symbols be denoted by *k*. Let g range over a set $\mathcal{G}$ of global definitions whose right-hand sides are all values. Recursion is only allowed at the top level but this restriction is not particularly burdensome – if a local function needs to be recursive, it can always be lambda-lifted (Johnsson 1985) to the top level.

We abbreviate a list of expressions $e_1 \ldots e_n$ as $\overline{e}$, and a list of variables $x_1 \ldots x_n$ as $\overline{x}$. A program is an expression with no free variables except those defined in $\mathcal{G}$. We let $[\overline{e}/\overline{x}]e'$ denote capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e'. We use $\equiv$ to denote equality up to renaming of variables.

The reduction semantics for the language is depicted in Figure 2.1. An evaluation context E is a term containing a single hole $[\,]$, which indicates the next expression to be reduced. The

$$
\begin{array}{lcll}
E\langle g\,\overline{e}\rangle & \mapsto & E\langle[\overline{e}/\overline{x}]e_f\rangle \text{ where } (g\,\overline{x} = e_f) \in \mathcal{G} & \text{(App)} \\
E\langle \textbf{case}\, k_j\,\overline{e}\,\textbf{of}\,\{p_i \to e_i\}\rangle & \mapsto & E\langle[\overline{e}/\overline{x}_j]e_j\rangle & \text{(Case)}
\end{array}
$$

Figure 2.1: Call-by-name reduction semantics

$$
\begin{array}{lcll}
\mathcal{D}[\![x]\!]_\rho & = & x & (1) \\
\mathcal{D}[\![k\,\overline{e}]\!]_\rho & = & k\,\mathcal{D}[\![\overline{e}]\!]_\rho & (2) \\
\mathcal{D}[\![E\langle g\,\overline{e}\rangle]\!]_\rho & = & h'\,\overline{y}, \text{ if } \exists (t,h') \in \rho\,.\,t \equiv E\langle g\,\overline{e}\rangle & (3) \\
 & & h\,\overline{y}, \text{ and } \mathcal{G}' := \mathcal{G}' \cup (h, e'), \text{ otherwise} & \\
 & & \text{where } h \text{ fresh}, \overline{y} = fv(E\langle g\,\overline{e}\rangle) & \\
 & & \text{and } e' = \mathcal{D}[\![E\langle[\overline{e}/\overline{x}]e_f\rangle]\!]_{\rho'} \text{ if } (g\,\overline{x} = e_f) \in \mathcal{G} & \\
 & & \text{and } \rho' = \rho \cup (E\langle g\,\overline{e}\rangle, h) & \\
\mathcal{D}[\![E\langle \textbf{case}\, x\,\textbf{of}\,\{p_i \to e_i\}\rangle]\!]_\rho & = & \textbf{case}\, x\,\textbf{of}\,\{p_i \to \mathcal{D}[\![E\langle e_i\rangle]\!]_\rho\} & (4) \\
\mathcal{D}[\![E\langle \textbf{case}\, k_j\,\overline{e}\,\textbf{of}\,\{p_i \to e_i\}\rangle]\!]_\rho & = & \mathcal{D}[\![E\langle[\overline{e}/\overline{x}_j]e_j\rangle]\!]_\rho & (5)
\end{array}
$$

Figure 2.2: First order deforestation

expression $E\langle e\rangle$ is the term obtained by replacing the hole in E with e. $\overline{E}$ denotes a list of terms with just a single hole, evaluated from left to right. The evaluation contexts are:

$$E \;::=\; [\,] \mid E\,\overline{e} \mid \textbf{case}\,E\,\textbf{of}\,\{p_i \to e_i\}$$

2.1 Deforestation

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

We denote the transformation algorithm $\mathcal{D}$ and let $\mathcal{D}[\![e]\!]_\rho$ denote the result of transforming an expression *e*. The memoization list ρ holds information about expressions the algorithm has already traversed. The transformation takes an expression and a set of global definitions as input, and returns a new expression together with a new set of global definitions. A simple first order deforestation that includes folding is shown in Figure 2.2.

Since the transformation can inline recursive functions (rule 3), there needs to be a mechanism for termination. For deforestation, it is usually accomplished by showing that the depth and width of terms are bounded and that the transformation only can decrease the depth of the term it is transforming.

The mechanism to bound the width of terms is the type system. The width of a term can only grow without bounds if a recursive function is supplied with more arguments than its type can allow. Since any well-typed term must have a finite type, the type system does not allow this. In the case of polymorphism, all particular instances have to have a finite type as well.

The bound on depth of terms is placed by the *treeless form*, and the treeless form has recieved more attention after Wadler's initial work. Hamilton (1993) extends the treeless form to permit more programs to be deforested.

$$
\begin{array}{lcll}
\mathcal{D}[\![x]\!]_\rho & = & x & (1)\\
\mathcal{D}[\![k\,\overline{e}]\!]_\rho & = & k\,\mathcal{D}[\![\overline{e}]\!]_\rho & (2)\\
\mathcal{D}[\![E\langle g\,\overline{e}\rangle]\!]_\rho & = & h'\,\overline{y}, \text{ if } \exists (t,h') \in \rho\,.\,t \equiv E\langle g\,\overline{e}\rangle & (3)\\
 & = & h\,\overline{y}, \text{ and } \mathcal{G}' := \mathcal{G}' \cup (h,e'), \text{ otherwise} & \\
 & & \text{where } h \text{ fresh, } \overline{y} = fv(E\langle g\,\overline{e}\rangle) & \\
 & & \text{and } e' = \mathcal{D}[\![E\langle[\overline{e}/\overline{x}]e_f\rangle]\!]_{\rho'} \text{ if } (g\,\overline{x} = e_f) \in \mathcal{G} & \\
 & & \text{and } \rho' = \rho \cup (E\langle g\,\overline{e}\rangle, h) & \\
\mathcal{D}[\![E\langle \textbf{case}\,x\,\textbf{of}\,\{p_i \to e_i\}\rangle]\!]_\rho & = & \textbf{case}\,x\,\textbf{of}\,\{p_i \to \mathcal{D}[\![[p_i/x]E\langle e_i\rangle]\!]_\rho\} & (4')\\
\mathcal{D}[\![E\langle \textbf{case}\,k_j\,\overline{e}\,\textbf{of}\,\{p_i \to e_i\}\rangle]\!]_\rho & = & \mathcal{D}[\![E\langle[\overline{e}/\overline{x}_j]e_j\rangle]\!]_\rho & (5)
\end{array}
$$

Figure 2.3: The positive supercompiler

Marlow and Wadler (1992) adressed the restriction to a first-order language when they presented a deforestation algorithm for a higher order language. This work was refined in Marlow's (1995) dissertation, where he also related deforestation to the cut-elimination principle of logic.

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

The work by Marlow is very close algorithmically to our positive supercompiler in Chapter 3, but was not included in the Glasgow Haskell Compiler (GHC) (GHC 2008) since short cut deforestation (Section 2.4.2) was a much simpler alternative that gave results that were good enough.

2.2 Supercompilation

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

The *positive supercompiler* (Sørensen et al. 1996) is a variant which only propagates positive information, such as equalities. The propagation is done by unification and the work highlights how similar deforestation and positive supercompilation really are. The rules for a simple first order positive supercompiler are shown in Figure 2.3. Rule 4 is the only rule that is changed from the deforestation algorithm in Figure 2.2.

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

More recently, Mitchell and Runciman (2008) have worked on supercompiling Haskell.

Table 2.1: Taxonomy of transformers

Transformer	Information Propagation	Eval. strategy	Data structure removal
Partial evaluation	Constant	CBV	No
Deforestation	Constant	CBN	Yes
Positive SCP	Unification	CBN	Yes
Perfect SCP	Constraint	CBN	Yes
GPC	Constraint	CBN	Yes

They report runtime reductions of up to 55% when their supercompiler is used in conjunction with GHC.

Glück and Sørensen (1996) have prepared a table with the taxonomy of transformers shown in Table 2.1. The evaluation strategy in column 3 is for the transformer in question, not the language it is applied to. Supercompilation for Refal applied a call-by-name transformation to a call-by-value language, thereby introducing termination in certain cases. While supercompilation has obvious strength benefits compared to deforestation, it has only been investigated by a small circle of experts and not gotten a foothold outside of that circle.

2.3 Generalized Partial Computation

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

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

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

We note that the rules for the first order language presented by Takano (1991) are very similar to the positive supercompiler, but the theorem prover required might exclude the technique as a candidate for automatic compiler optimisations. The lack of termination guarantees for the transformation might be another obstacle.

2.4 Other transformations

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

2.4.1 Partial Evaluation

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

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

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

2.4.2 Short Cut Deforestation

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

The idea is that the constructors *Nil* and *Cons* can be replaced by a *foldr* consumer, and a special function *build* is used for the transformation to recognize the producer and enforce the type requirement.

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

Foldr can be defined as follows:

$$\begin{aligned} &foldr\ f\ c\ [] &&= c \\ &foldr\ f\ c\ (x : xs) &&= f\ x\ (foldr\ f\ c\ xs) \end{aligned}$$

Foldr is quite expressive (Hutton 1999) and many list consuming functions can be written using it. The special function *build* requires rank-n-polymorphism (Odersky and Läufer 1996), and is defined as:

$$\begin{aligned} &build &&:: (\forall\, b.\, (a \to b \to b) \to b \to b) \to [a] \\ &build\ g &&= g\ (:)\ [] \end{aligned}$$

Lists using *build/foldr* can easily be removed with the *foldr/build* rule, whose correctness comes from the free theorems (Wadler 1989):

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

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

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

2.4.3 Type-inference Based Short Cut Deforestation

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

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

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

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

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

2.4.4 Zip fusion

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

It can remove all argument lists from a function which consumes more than one list. The method described by Svenningsson will remove all intermediate lists in $zip\ [1..n]\ [1..n]$, one of the main criticisms against the *foldr/build* rule.

The technique can remove intermediate lists from functions which consume their lists using accumulating parameters, a known problematic case when fusing functions that most techniques can not handle. The *destroy/unfoldr* rule is defined as:

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

Just like in short cut deforestation, this rule is not valid for any g, only those of the type:

$$g\ ::\ \forall\ a.(a\ \rightarrow\ Maybe\ (b, a))\ \rightarrow\ a\ \rightarrow\ c$$

This can be ensured by giving *destroy* the type:

$$destroy\ ::\ (\forall\ a.(a\ \rightarrow\ Maybe\ (b, a))\ \rightarrow\ a\ \rightarrow\ c)\ \rightarrow\ [b]\ \rightarrow\ c$$

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

2.4.5 Lightweight Fusion

Ohori and Sasano (2007) presents a lightweight fusion algorithm which works by promoting a fix point. The transformation is implemented in a variant of a compiler for Standard ML and some benchmarks are presented. Correctness is proven for a call-by-name language, but their implementation is for a call-by-value language. The algorithm is restricted to inline each function at most once, and avoids any problems with termination that way. This gives the unfortunate side effect that it does not handle two successive applications of the same function, nor mutually recursive functions.

Considering the early stage of their work, it is an interesting approach that seems to solve a lot of problems. It is explicitly mentioned that their goal is to extend the transformation to work for an impure call-by-value functional language.

CHAPTER 3

Higher order Supercompilation

> Was man nicht versteht,
> besitzt man nicht.
> – *Goethe*

An algorithm that specializes programs and automatically removes intermediate data structures for a higher-order call-by-value language is presented. It is proven correct, and important design issues for the algorithm are presented. An edited version of this Chapter appeared in the draft proceedings of IFL'07 (Jonsson and Nordlander 2007).

We want the compiler to perform program specialization, remove intermediate structures and preserve semantics. This allows the programmer to write concise and elegant programs without concern for implementation efficiency. Many of the algorithms described in Chapter 2 either perform program specialization or intermediate structure removal, but not both. Possible candidates are the transformations directly related to ordinary deforestation, but no such algorithm exists with call-by-value semantics. Our choice of supercompilation rests on the belief that it gives a good power to weight ratio – being a stronger transformation than deforestation but not depending on a theorem prover like GPC does.

3.1 Motivation and Examples

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

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

If we naïvely change Wadler's algorithm to call-by-value semantics by eagerly attempting to transform arguments before attacking the body, we do not achieve this improvement in

complexity. An example from a hypothetical deforestation algorithm that attacks arguments first is:

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

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

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

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

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

(The expression contains a previously seen expression.
$x : append\ xs\ ys'$ is extracted for separate transformation)

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

(A renaming of a previous expression in the (x':xs') branch)

The end result from this transformation is:

$$\begin{array}{l}\qquad\qquad\textbf{case}\ xs'\ \textbf{of}\\ \qquad\qquad\quad []\ \rightarrow\ h_1\ ys'\ zs'\\ \qquad\qquad\quad (x : xs)\ \rightarrow\ h_1\ (h_2\ x\ xs\ ys')\ zs'\\ h_1\ xs\ ys\quad = \textbf{case}\ xs\ \textbf{of}\ \{[]\ \rightarrow\ ys; (x' : xs')\ \rightarrow\ x' : h_1\ xs'\ ys\}\\ h_2\ x\ xs\ ys = x : \textbf{case}\ xs\ \textbf{of}\ \{[]\ \rightarrow\ ys; (x' : xs')\ \rightarrow\ h_2\ x'\ xs'\ ys\}\end{array}$$

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

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

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

$$
\begin{array}{l}
square\ x\ =\ x * x \\
map\ f\ xs = \textbf{case}\ xs\ \textbf{of}\ \{[]\ \rightarrow\ []\mathrm{;}\ (x' : xs')\ \rightarrow\ (f\ x') : (map\ f\ xs')\ \} \\
sum\ xs\ \ \ = \textbf{case}\ xs\ \textbf{of}\ \{[]\ \rightarrow\ 0\mathrm{;}\ (x' : xs')\ \rightarrow\ x'\ +\ sum\ xs'\ \}
\end{array}
$$

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

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

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

$$
\begin{array}{l}
\textbf{case}\ (\textbf{case}\ ys\ \textbf{of}\ \{[]\ \rightarrow\ []\mathrm{;}\ (x' : xs')\ \rightarrow\ (square\ x') : (map\ square\ xs')\})\ \textbf{of} \\
\quad []\ \rightarrow\ 0 \\
\quad (x' : xs')\ \rightarrow\ x'\ +\ sum\ xs'
\end{array}
$$

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

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

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

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

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

The following examples are due to Ohori and Sasano (2007). We need the following new function definitions:

$$
\begin{array}{l}
mapsq\ xs = \textbf{case}\ xs\ \textbf{of}\ \{\ []\ \rightarrow\ []\mathrm{;}\ (x' : xs')\ \rightarrow\ (x' * x') : (mapsq\ xs')\ \} \\
f\ xs\ \ \ \ \ \ = \textbf{case}\ xs\ \textbf{of}\ \{\ []\ \rightarrow\ []\mathrm{;}\ (x' : xs')\ \rightarrow\ (2 * x') : (g\ xs')\ \ \ \ \ \ \\
g\ xs\ \ \ \ \ \ = \textbf{case}\ xs\ \textbf{of}\ \{\ []\ \rightarrow\ []\mathrm{;}\ (x' : xs')\ \rightarrow\ (3 * x') : (f\ xs')\ \ \ \ \ \ \\
\end{array}
$$

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

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

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

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

This will fold in a few steps, and the final result of our transformation is $h_1\ xs$, with the new residual function h_1 that only traverses its input once:

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

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:

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

Moving down the outer case into each branch, performing reductions to end up with:

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

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

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

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

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

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

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

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

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

Expressions

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

Values

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

Figure 3.1: The language

This is simplified by our positive supercompiler to:

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

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

3.2 Language

Our language of study is a strict, higher-order, functional language with let-bindings and case-expressions. Its syntax is presented in Figure 3.1. This syntax should be close to an intermediate format in a compiler for a functional language.

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 $\oplus$. All functions have a specific arity and all applications must be saturated; hence the expression $\lambda x.map\ (\lambda\ y.y + 1)\ x$ is legal whereas $map\ (\lambda\ y.y + 1)$ is not. The intended operational semantics is given in Figure 3.2. The language is supposed to be typed using the Hindley-Milner type system (Damas and Milner 1982).

We denote the free variables of an expression *e* by $fv(e)$, as defined in Figure 3.3. Along the same lines we denote the function names in an expression *e* as $fn(e)$, defined in Figure 3.4.

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

Reduction contexts

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

Evaluation relation

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

Figure 3.2: Reduction semantics

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

Figure 3.3: Free variables of an expression

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

Figure 3.4: Function names of an expression

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

3.3 Higher Order Positive Supercompilation

Our supercompiler is defined as a set of rewrite rules that pattern-match on expressions. This algorithm is called the *driving* algorithm, and is defined in Figure 3.5. Two additional parameters appear as subscripts to the rewrite rules: a memoization list ρ and a driving context $\mathcal{R}$. The memoization list holds information about expressions already traversed and is explained more in detail in Section 3.3.1. The driving context $\mathcal{R}$ is smaller than $\mathcal{E}$, and is defined as follows:

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

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

In principle, an expression e is strict with regards to a variable x if it eventually evaluates x; in other words, if $e \mapsto \ldots \mapsto \mathcal{E}\langle x \rangle$. Such information is in general not computable, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 3.6, where the strict variables of an expression e are defined as all free variables of e except those that only appear under a lambda or not inside all branches of a case. The function *strict* is a mapping from expressions to multisets.

If a variable appears no more than once in a term, that term is said to be *linear* with respect to that variable. Like Wadler (1990), we extend the definition slightly for linear **case** terms: no variable may appear in both the selector and a branch, although a variable may appear in more than one branch. The definition of *append* is linear, although ys appears in both branches. An approximation of *linear* is that there exists one and only one occurence of the variable in the multiset returned from *strict*, which we denote by exists with an exclamation mark, $\exists! x \in \textit{strict}(e)$.

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R11 and R20 are the default fallback cases which extend the given driving context $\mathcal{R}$ and zoom in on the next expression to be driven. The program is turned "inside-out" by moving the surrounding context $\mathcal{R}$ into all branches of the case-statement through rules R16 and R19. Rules R14 and R15 have a similar mechanism for let-statements and letrec-statements. Notice how the context is moved out of the recursive call in rule R8, whereas rule R7 recursively applies the driving algorithm to the full new term $\mathcal{R}\langle n \rangle$, forcing a re-traversal of the new term in hope of further reductions. Meta-variable a in rule R19 stands for an "annoying" expression; expressions that might cause our supercompiler to not be recursively applied to the full term. The grammar for annoying expressions is:

$$a \;::=\; x \mid e \oplus f \mid a\,\overline{e}$$

$$
\begin{array}{lcll}
\mathcal{D}[\![n]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle n\rangle & \text{(R1)}\\
\mathcal{D}[\![x]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle x\rangle & \text{(R2)}\\
\mathcal{D}[\![g]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}_{app}(g,)_{\mathcal{R},\mathcal{G},\rho} & \text{(R3)}\\
\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho} & = & k\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho} & \text{(R4)}\\
\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle & \text{(R5)}\\
\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{G},\rho} & = & (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}) & \text{(R6)}\\
\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho}\text{, where } n = n_1 + n_2 & \text{(R7)}\\
\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho} \oplus \mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}\rangle & \text{(R8)}\\
\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & \text{(R9)}\\
\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho} & \text{(R10)}\\
\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,\overline{e}\rangle,\mathcal{G},\rho} & \text{(R11)}\\
\mathcal{D}[\![\textbf{let}\,x = v\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R12)}\\
\mathcal{D}[\![\textbf{let}\,x = y\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R13)}\\
\mathcal{D}[\![\textbf{let}\,x = e\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{[],\mathcal{G},\rho}\text{, if } \exists!x \in \mathit{strict}(f) & \text{(R14)}\\
 & & \textbf{let}\,x = \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}\,\textbf{in}\,\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}\text{, otherwise} & \\
\mathcal{D}[\![\textbf{letrec}\,g = v\,\textbf{in}\,e]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{letrec}\,g = v\,\textbf{in}\,\mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{[],\mathcal{G}',\rho} & \text{(R15)}\\
 & & \text{where } \mathcal{G}' = \mathcal{G} \cup (g,v) & \\
\mathcal{D}[\![\textbf{case}\,x\ \textbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{case}\,x\ \textbf{of}\,\{p_i \to \mathcal{D}[\![\mathcal{R}\langle[p_i/x]e_i\rangle]\!]_{[],\mathcal{G},\rho}\} & \text{(R16)}\\
\mathcal{D}[\![\textbf{case}\,k_j\,\overline{e}\,\textbf{of}\,\{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let}\,\overline{x}_j = \overline{e}\,\textbf{in}\,e_j]\!]_{\mathcal{R},\mathcal{G},\rho} & \text{(R17)}\\
\mathcal{D}[\![\textbf{case}\,n_j\ \textbf{of}\,\{n_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R18)}\\
\mathcal{D}[\![\textbf{case}\,a\ \textbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{case}\,\mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho}\,\textbf{of}\,\{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\} & \text{(R19)}\\
\mathcal{D}[\![\textbf{case}\,e\ \textbf{of}\,\{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle\textbf{case}\,[]\,\textbf{of}\,\{p_i\to e_i\}\rangle,\mathcal{G},\rho} & \text{(R20)}
\end{array}
$$

Figure 3.5: Driving algorithm

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

Figure 3.6: The strict variables of an expression

$$
\begin{array}{lcll}
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & h'\,\overline{x}, & \text{if } \exists (h',t) \in \rho\,.\,t \equiv \mathcal{R}\langle g\,\overline{e}\rangle \\
\quad \text{where } \overline{x} = \text{fv}(\mathcal{R}\langle g\,\overline{e}\rangle) & & & \\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle g\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle, & \text{if } \exists (h',t) \in \rho\,.\,t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{letrec}\, h = \lambda\overline{x}.e'\, \textbf{in}\, h\,\overline{x}, & \text{if } h \in \mathit{fn}(e') \\
 & & e', & \text{otherwise} \\
\multicolumn{4}{l}{\quad \text{where } \overline{x} = \text{fv}(\mathcal{R}\langle g\,\overline{e}\rangle),\ \rho' = \rho \cup (h, \mathcal{R}\langle g\,\overline{e}\rangle),\ h \text{ fresh},} \\
\multicolumn{4}{l}{\qquad (g = v) \in \mathcal{G} \text{ and } e' = \mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}}
\end{array}
$$

Figure 3.7: Driving applications

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 R17).

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

The algorithm might leave unused function definitions in the program through rule R15, these can be removed in a separate pass after the transformation is done.

Notice that the primitive operations ranged over by $\oplus$ in this language are supposed to be strict with regards to both arguments. This stands in contrast to ordinary functions, which can be inlined and partially evaluated for a couple of steps even if the arguments are unknown. Our algorithm leaves a primitive operation in place if any of its arguments is not a primitive value.

If we had a perfect strictness analysis and could decide whether an arbitrary expression will terminate or not, the only difference in results between our algorithm and a call-by-name counterpart would be for the non-terminating cases. In practice, we have to settle for an approximation, such as the simple analysis defined in Figure 3.6. One may speculate whether the transformations thus missed will have adverse effects on the usefulness of our algorithm in practice. We believe we have seen clear indications that this is not the case, and that crucial factor instead is the ability to inline function bodies irrespective of whether arguments are values or not.

3.3.1 Application Rule

In the driving algorithm rule R3 and rule R9 refers to $\mathcal{D}_{app}(\,)$, defined in Figure 3.7. $\mathcal{D}_{app}(\,)$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation.

Care needs to be taken to ensure that recursive functions are not inlined forever. The driving algorithm keeps a record of previously seen applications in the memoization list ρ; whenever it detects an expression that is equivalent (up to alpha conversion) to a previous expression, the algorithm creates a new recursive function h_n for some n. Whenever such an expression is encountered again, a call to h_n is inserted. This is not sufficient to guarantee termination of the algorithm, but the mechanism is crucial for the complexity improvements mentioned in Section 3.1.

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

We need a definition of uniform terms analogous to the work by Sørensen and Glück (1995), with some small adjustments specific to our language.

Definition 3.2 (Uniform terms)**.** *Let s range over the set* $N \cup X \cup K \cup \{\mathbf{caseof}, \mathbf{letin}, \mathbf{letrec}, \mathbf{primop}, \mathbf{lambda}, \mathbf{apply}\}$*, and let* $\mathbf{caseof}(\overline{e}), \mathbf{letin}(\overline{e}), \mathbf{letrec}(\overline{v}, e), \mathbf{primop}(\overline{e}), \mathbf{lambda}(e)$*, and* $\mathbf{apply}(\overline{e})$ *denote a case, let, recursive let, primitive operation, lambda abstraction or application containing subexpressions* $\overline{e}, e$ *or* $\overline{v}$*. The set of terms T is the smallest set of arity respecting symbol applications* $s(\overline{e})$*.*

Definition 3.3 (Homeomorphic embedding)**.** *Define* $\trianglelefteq$ *as the smallest relation on* T *satisfying:*

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

3.3.2 Termination

In order to prove that the algorithm terminates we show that each recursive application of $\mathcal{D}[\![\ ]\!]$ in the right-hand sides of definition 3.5 and 3.7 has a strictly smaller weight than the left-hand side. The weight of an expression is one plus the sum of the weight of its subexpressions, and the weight of the entire transformation is a triple that contains the maximum length of the memoization list ρ, the weight of the term being transformed and the weight of the current term in focus.

Definition 3.4. *The weight of an expression is* $|s(e_1, \ldots, e_n)| = 1 + \sum_{i=1}^{n} |e_i|$.

Definition 3.5. *Let S be a set with a relation* $\leq$*. Then* $(S, \leq)$ *is a quasi-order if* $\leq$ *is reflexive and transitive.*

Definition 3.6. *Let* $(S, \leq)$ *be a quasi-order.* $(S, \leq)$ *is a well-quasi-order if, for every infinite sequence* $s_0, s_1, \ldots \in S$*, there are* $i < j$ *with* $s_i \leq s_j$

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

Lemma 3.7 (Higman's lemma)**.** *If a set* S *is well-quasi-ordered, then the set* S^* *of finite sequences over* S *is well-quasi-ordered.*

To prove termination we need a theorem from Dershowitz (1987), known as Kruskal's tree theorem:

Theorem 3.8 (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 (Simliar to Dershowitz (1987)). Collapse all integers to a single 0-ary constructor, and all variables to a different 0-ary constructor. Only applications of the same arity are considered equal and only applications of finite arity are included. A 3-ary application is different from a 4-ary application.

Suppose the theorem were false. Let the infinite sequence $\overline{t} = t_1, t_2, \ldots$ of terms be a minimal counterexample, measured by the size of the t_i. By the minimality hypothesis, the set of proper subterms of the t_i must be well-quasi-ordered, or else there would be a smaller counterexample $t_1, t_2, \ldots, t_{l-1}, s_1, s_2, \ldots$, where $s_1, s_2, \ldots$ is a counterexample of proper subterms, such that s_1 is a subterm of some t_l and all s_i in the counterexample are subterms of one of $t_l, t_{l+1}, \ldots$. (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 embeds 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 3.7 that the set of finite sequences consisting of the immediate subterms of the elements in 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. □

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

Lemma 3.9. *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. By inspection of the rules it is clear that only rules R3/R9 introduces such new terms. Inspection of the RHS of $\mathcal{D}_{app}(),_{\mathcal{G},\rho}$:

- $h'\,\overline{x}$: No recursive application in the RHS.
- $\mathcal{R}\langle g\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle$: No modification of ρ and no introduction of new terms.
- **letrec** $h = \lambda\overline{x}.\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}$ **in** $h\,\overline{x}$: 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.
- $\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}$: No new terms are created, and the memoization list, ρ, is extended with terms from T.

□

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 3.10. *For a fix set of function symbols $\mathcal{G}$ it follows from Kruskal's Tree Theorem 3.8 that no infinite chains of terms can appear, so there exists a largest length of these chains N.*

We define the weight of driving a term as:

Definition 3.11. *The weight of the driving algorithm* $|\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle e\rangle|, |e|)$

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

Definition 3.12. *The order between two tuples* (n_1, n_2, n_3) *and* (m_1, m_2, m_3) *is:*

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

With these definitions in place, we can formulate a lemma and prove that the weight is decreasing in each step of our algorithm. The full proof is included in Appendix A.1.

Lemma 3.13. *For each rule R:* $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho} = e_1$ *in Definition 3.5 and 3.7 and each recursive application* $\mathcal{D}[\![e']\!]_{\mathcal{R}',\mathcal{G},\rho'}$ *in* e_1, $|\mathcal{D}[\![e']\!]_{\mathcal{R}',\mathcal{G},\rho'}| < |\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}|$

The driving algorithm must be well defined for all inputs, hence a lemma about totality of the algorithm is needed. The proof is included in Appendix A.2.

Lemma 3.14 (Totality)**.** *For all well-typed expressions* e, $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}$ *is matched by a unique rule in Figure 3.5.*

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

Proposition 3.15 (Termination)**.** *The driving algorithm* $\mathcal{D}[\![\ ]\!]$ *terminates for all well-typed inputs.*

Proof. Corollary 3.10 states that the weight of the transformation is defined. Lemma 3.9 guarantees that the memoization list ρ only contains terms from the initial input. By Lemma 3.13 the weight of the transformation decreases for each step and by Lemma 3.14 we know that each recursive application will match a rule.

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

3.3.3 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 alter whether a program terminates or not.

Sands (1996a) shows how a transformation can alter 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 $f\ 0$ 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 indeed preserved. We therefore define the standard notions of operational approximation and equivalence. A general context C which has zero or more holes in the place of some subexpressions is introduced.

Definition 3.16 (Operational approximation)**.** *The expression e operationally approximates e', $e \sqsubseteq e'$, if for all contexts C such that C[e], C[e'] are closed, if evaluation of C[e] terminates then so does evaluation of C[e'].*

Definition 3.17 (Operational equivalence)**.** *The expression e is operationally equivalent to e', $e \cong e'$, if $e \sqsubseteq e'$ and $e' \sqsubseteq e$*

The correctness of deforestation in a call-by-name setting has previously been shown by Sands (1996a) using his improvement theory. We use Sands's definitions for improvement and strong improvement:

Definition 3.18 (Improvement)**.** *The expression 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.*

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

Definition 3.19 (Strong Improvement)**.** *The expression e is strongly improved by e', $e \trianglerighteq_s e'$, iff $e \trianglerighteq e'$ and $e \cong e'$.*

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

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

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

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

Theorem 3.22 (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 (Sands 1996a) associated with unfold-fold transformations

Theorem 3.23 (Partial Correctness)**.** *If $e_i \cong e'_i$ for all $i \in I$ then $h_i \sqsubseteq g_i$, $i \in I$.*

which we combine with Theorem 3.21 to get total correctness for a transformation:

Corollary 3.24. *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 (Sands 1997), 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 the theorems by Sands to our language.

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

Lemma 3.25 (Sands (1996b))**.** *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 $e \sqsubseteq e'$.*

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

Lemma 3.26 (Sands (1996a))**.** *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 for total correctness. Since strong improvement is improvement in one direction and operational approximation in the other direction a set of approximation laws that correspond to the improvement laws in Lemma 3.25 is necessary.

Lemma 3.27. *Approximation laws*

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

Combining Lemma 3.25 and Lemma 3.27 gives us the final tools we need to prove strong improvement:

Lemma 3.28. *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* $e \trianglerighteq_s e'$.

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

We need to relate local recursion expressed using fix, and the recusive definitions which the improvement theorem is defined for. This is solved by a technical lemma that relates the cost of terms on a certain form to their recursive counterparts.

Proposition 3.29. *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 (1996a)). 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 3.26 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 3.22, we have a cost-equivalent transformation from h^- to h, where $h = [h/h^-][\lambda n.h^-\,n/g]e = [\lambda n.h\,n/g]e$.

□

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

Proposition 3.30. *Consequences of the letrec definition*

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

ii) $\textbf{letrec}\,h = \lambda\overline{x}.e\,\textbf{in}\,h \trianglelefteq\!\trianglerighteq \lambda n.fix\,(\lambda h.\lambda\overline{x}.e)\,n$

iii) $\textbf{letrec}\,h = \lambda\overline{x}.e\,\textbf{in}\,e' \trianglelefteq\!\trianglerighteq [\textbf{letrec}\,h = \lambda\overline{x}.e\,\textbf{in}\,h/h]e'$

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). □

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

Theorem 3.31 (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 3.29 $g \trianglelefteq\!\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_0)$. Use this, the congruence properties, and the properties listed in Proposition 3.30 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 3.24, $g \trianglerighteq_s g'$ 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 3.29, $g' \trianglelefteq\!\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_1)$. 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 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 3.30 we get $\textbf{letrec}\, h = \lambda\overline{x}.e_0\, \textbf{in}\, h \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_0 1\, \textbf{in}\, h$, which can be further expanded by congruency properties of improvement to $[\textbf{letrec}\, h = \lambda\overline{x}.e_0\, \textbf{in}\, h/h]e \trianglerighteq_s [\textbf{letrec}\, h = \lambda\overline{x}.e_0 1\, \textbf{in}\, h/h]e$. Using Proposition 3.30 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 proposition. □

Proposition 3.32 (Total Correctness)**.** *Let e be an expression, and ρ an environment such that*

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

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

The full proof is included in Appendix A.4, but the proof does not reveal anything unexpected. The proof for rule R9 is similar in structure to the proof by Sands (1996a, p. 24).

3.4 Positive Supercompilation versus Deforestation

The additional strength in positive supercompilation compared to deforestation comes from propagating information about the pattern of each branch in a case statement (rule R16 in our definition). It is not hard to craft a program by hand that benefits from this extra information propagation. It turns out that this information is useful for real programs as well. An example collected from Conway's Life in the nofib benchmark suite (Partain 1992) is the expression $concat\ (map\ star\ xs)$ with star defined as:

$$star\ n = \ \textbf{case}\ n\ \textbf{of}\ \{0\ \rightarrow\ ''\,'';\ 1\ \rightarrow\ ''o''\}$$

Evaluating $\mathcal{D}[\![\ concat\ (map\ star\ xs)]\!]$ results in the function $h_1'\ xs$:

$$\begin{array}{l}
h_1'\ (x : xs) = \ \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad 0\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \ \rightarrow\ [' \,']\\
\qquad\qquad\qquad\qquad (x' : xs') \ \rightarrow\ \textbf{let}\ h_2\ =\ h_1'\ xs\ \textbf{in}\ [' \,',\ h_2] \\
\qquad\qquad 1\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \ \rightarrow\ [' \,',' \,o'])\\
\qquad\qquad\qquad\qquad (x' : xs') \ \rightarrow\ \textbf{let}\ h_3\ =\ h_1'\ xs\ \textbf{in}\ [' \,',\ 'o',\ h_3] \\
h_1\ (x : xs) = \ \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad 0\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \ \rightarrow\ \textbf{case}\ x\ \textbf{of}\ \{\ 0\ \rightarrow\ [' \,'];\ 1\ \rightarrow\ [' \,',\ 'o']\ \}\\
\qquad\qquad\qquad\qquad (x' : xs') \ \rightarrow\ \textbf{let}\ h_2\ =\ h_1\ xs\ \textbf{in case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 0\ \rightarrow\ [' \,',\ h_2] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1\ \rightarrow\ [' \,',\ 'o',\ h_2] \\
\qquad\qquad 1\ \rightarrow\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \ \rightarrow\ \textbf{case}\ x\ \textbf{of}\ \{\ 0\ \rightarrow\ [' \,'];\ 1\ \rightarrow\ [' \,',\ 'o']\ \}\\
\qquad\qquad\qquad\qquad (x' : xs') \ \rightarrow\ \textbf{let}\ h_3\ =\ h_1\ xs\ \textbf{in case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 0\ \rightarrow\ [' \,',\ h_3] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1\ \rightarrow\ [' \,',\ 'o',\ h_3]
\end{array}$$

In comparison, the function h_1 is the output from our algorithm if we remove the extra information propagation from rule R16. There are two case-statements in this function that pattern matches on x. The second one is not necessary, since the value of x must be either 0 or 1 depending on which branch it has previously taken.

3.5 Extended Let-rule

The driving algorithm can be extended in various ways that will make it more powerful. We show that with an extended Let-rule in combination with a disabled whistle for closed expressions we can evaluate many closed expressions.

If the let-expression contains no free variables we could drive either the right hand side or the body and see what the result is. We could augment rule R14 with a second and third

alternative:

$$\begin{array}{lll}
\mathcal{D}[\![\textbf{let}\ x = e\ \textbf{in}\ f]\!]_{\mathcal{R},\mathcal{G},\rho} = & \mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho}, & \text{if } \exists!x \in \textit{strict}(f) \\
& \mathcal{D}[\![\mathcal{R}\langle [v/x]f\rangle]\!]_{[],\mathcal{G},\rho}, & \text{if } \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho} = v \\
& \mathcal{D}[\![\mathcal{R}\langle [e/x]f'\rangle]\!]_{[],\mathcal{G},\rho}, & \text{if } \mathcal{D}[\![f]\!]_{[],\mathcal{G},\rho} = f', \\
& & \text{and } \exists!x \in \textit{strict}(f') \\
& \textbf{let}\ x = \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}\ \textbf{in}\ \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}, & \text{otherwise}
\end{array}$$

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

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

$$bodyEx = \textbf{let}\ x\ =\ e\ \textbf{in case}\ []\ \{[]\ \rightarrow\ x;\ (x : xs)\ \rightarrow\ y\}$$

$$\begin{array}{lll}
 & \mathcal{D}[\![\ bodyEx\ ]\!] & \\
= & \mathcal{D}[\![\ \textbf{let}\ x\ =\ e\ \textbf{in case}\ []\ \textbf{of}\ \{\ []\ \rightarrow\ x;\ (x : xs)\ \rightarrow\ y\ \}\ ]\!] & \text{(R9)} \\
= & \mathcal{D}[\![\ e\ ]\!] & \text{(R14)}
\end{array}$$

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

Duplicating code to enable further transformations might sometimes be beneficial. Removing the linearity constraint of rule R14 could enable further transformations, but it could just as well turn out to just having duplicated work that forces multiple evaluation of the "same" expression, as well as growth in code size. The current algorithm has no means of finding a suitable trade-off, so it ignores the problem and always performs substitution when strictness so allows. Obtaining a more refined behavior in this respect is left for future work.

If rule R14 is not restricted by the linearity constraint, it is not an improvement, and our proof of correctness is no longer valid. Two concerns need to be addressed to ensure its correctness: (i) altering of termination properties, and (ii) final value of computation.

Compare the two expressions $\textbf{let}\ x = e\ \textbf{in}\ f$ and $[e/x]f$, where *f* is strict with respect to *x*. By assumption, x is unique so there is no name capture, hence *x* can not be bound in $\mathcal{R}$. This allows us to leave out the surrounding contexts $\mathcal{R}$ in our reasoning.

Regardless of whether *e* terminates or not it will be evaluated eventually due to our definition of strictness: $f \mapsto \ldots \mapsto \mathcal{E}\langle x\rangle$, hence $[e/x]f \mapsto \ldots \mapsto \mathcal{E}\langle e\rangle$, where *e* is our next redex. Whatever termination behaviour e had in $\textbf{let}\ x = e\ \textbf{in}\ f$ will remain in the evaluation of $\mathcal{E}\langle e\rangle$.

If *f* is non-linear but strict with respect to *x*, we might introduce work duplication by performing the substitution. While this is bad from an efficiency viewpoint, it is orthogonal to the issue of preservation of semantics. Since there are no side effects in our language and our evaluation relation is deterministic, *e* will evaluate to the same value *v* every time, regardless of surrounding context.

Based on this informal argumentation we claim that while our extended rule R14 is not a strong improvement, operational equivalence still holds and it is therefore correct.

3.6 Improved Driving of Applications

One of the most important parts of the algorithm is the folding mechanism. Without the folding mechanism, no removal of intermediate data structures will occur. We suggest an improved $\mathcal{D}_{app}(\,)$ which is fairly complex, but enables more fold opportunities.

Whenever the whistle blows, our improved algorithm splits the current 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 previous algorithm would just stop in these cases, leaving any intermediate structures in place. The design follows the positive supercompilation as outlined by Sørensen (2000), except that we need to substitute the transformed expressions back instead of pulling them out into let-statements, in order to preserve strictness. Our algorithm is also more complicated because we perform the program extraction immediately instead of constructing a large tree and extracting the program in a separate pass.

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

Definition 3.33 (Most specific generalization)**.**

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

We refer to t_g as the ground term. For background information and an algorithm to compute most specific generalizations, see Lassez et al. (1988). An example of the homeomorphic embedding and the msg is:

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

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

Definition 3.34 (Split)**.** *For* $t \in T$ *we define* $split(t)$ *by:*

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

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

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

Figure 3.8: Improved driving of applications

$$
\begin{array}{lcl}
GEN(e_1, e_2) & = & (\theta t, \textit{used ++ concat used'}) \\
\multicolumn{3}{l}{\quad \text{where } (t_g, \theta_1', \theta_2') = msg(e_1, e_2),\ (t', \sigma) = split\,\theta_1'} \\
(t, used) & = & \mathcal{D}[\![t_g]\!]_{[],\mathcal{G},\rho}, \text{ if } t_g \neq x \\
 & & \mathcal{D}[\![t']\!]_{[],\mathcal{G},\rho}, \text{ otherwise} \\
(\theta, used') & = & \text{unzip } \mathcal{D}[\![\theta_1']\!]_{[],\mathcal{G},\rho}, \text{ if } t_g \neq x \\
 & & \text{unzip } \mathcal{D}[\![\sigma]\!]_{[],\mathcal{G},\rho}, \text{ otherwise}
\end{array}
$$

Figure 3.9: Generalization

The new $\mathcal{D}_{app}(\,)$ outlined in Figure 3.8 is fairly complex. We need to be able to generalize both upwards and downwards, and hence information must be propagated in both directions. This requires changing return type to a pair, which in turn requires glue code for all the rules in Figure 3.5 that will merge the second component of the return values. An example from a Haskell implementation:

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

Alternative 2 of $\mathcal{D}_{app}(\,)$ is for upwards generalization, and alternatives 3 and 4 are for the downwards generalization. The generalization algorithm is shown in Figure 3.9. If the ground term t_g is a variable the algorithm drives the output from split, otherwise it will drive the output from the msg.

3.7 Classic Compiler Optimizations

Supercompilation provides a framework that allows some of the classical compiler optmizations to be enabled without much effort.

3.7.1 Constant Folding

Constant folding is evaluation of expressions that have constant operands at compile time. Simple integer arithmetic like $3 + 4$ can be replaced by its value 7. Performing floating point operations is a bit more tricky, since the compiler writer has to ensure that the floating point on the compiling platform behaves the same as on the target platform. If there is a mismatch, code could produce different results depending on if the calculations are performed by the compiler or the target program! In a functional setting like ours, it is easy to determine whether a parameter is a value, or something else.

3.7.2 Copy Propagation

Copy propagation takes assignments of variables, $x = y$, and replaces later uses of x with y, as long as the value of neither x nor y are changed. Most languages have some way to give a second name, an alias, to the storage area pointed to by a variable. It is therefore difficult to determine whether the value of a variable has changed in general.

However, in our context the value of variables can never change. If we have a let expression where the right hand side is a variable, we can simply substitute that variable everywhere in the body without changing semantics, hopefully reducing register pressure and space requirements. We see no reason for not enabling this simple optimization.

3.7.3 Dead Code Elimination

Dead code might also be called unreachable code. This is code that can not possibly be reached and executed under any circumstances, regardless of input data. Constructing a call-graph for the program and removing any code that has no edges to it is a simple form of dead code elimination. Several classical compiler optimizations produce dead code, leaving it to other transformations to clean up after them.

In our setting, a let expression where the right hand side is an expression that terminates, and the body does not contain the bound variable, we can safely throw away the expression and remove the variable binding.

3.8 Comparison to previous work

3.8.1 Call-by-value versus Call-by-name

Deforestation, Supercompilation and GPC as presented in Chapter 2 are all transformations that have call-by-name semantics. This is a significant difference to the positive supercompiler as outlined in this chapter, which has call-by-value semantics.

Since we are restricted to call-by-value semantics we sometimes have to keep expressions that the termination analysis decides might be non-terminating. Our termination analysis has to be conservative for our positive supercompiler to be correct, some terminating expressions will be considered non-terminating.

As previously stated, the non-terminating expressions are not very interesting in reality, since pracitcal programs will contain very few of them.

3.8.2 Taxonomy of transformers

Important characteristics of the positive supercompiler with call-by-value semantics, using the same criteria as Table 2.1:

Information Propagation Our positive supercompiler propagates more information than deforestation, but less than the perfect supercompiler by Secher (1999). This puts it somewhere in the middle of the scale. There is no inherent reason why the information propagation must remain as it is, converting it to propagating constraints should be straight forward.

Transformation Strategy For transformation strategy, we use call-by-name where expressions are strict, which is no different from the other call-by-name transformations. The difference between our positive supercompiler and the one by Sørensen et al. (1996) is when expressions are not strict, and we have to obey the call-by-value semantics.

It is important to notice that it is still possible to supercompile the bodies of functions, even when they are not strict and their parameters are not values – the supercompiler can never get stuck and be unable to proceed.

Data structure removal Data structure removal is performed by the positive supercompiler as well.

The comparison with transformations that have call-by-name semantics is not that straightforward. Since call-by-name terminates more often than call-by-value one could argue that even the simplest call-by-name transformation mentioned, deforestation (Wadler 1990), is more powerful than any transformation with call-by-value semantics.

Our positive supercompiler propagates positive information into the branches of case-statements, has an evaluation strategy that is call-by-name if the input is terminating and does remove intermediate data structures. We demonstrated a real example of where the information propagation was useful in Section 3.4.

On the grounds that non-terminating functions are not very interesting for real world applications the positive supercompiler for call-by-value is placed right above its sibling the positive supercompiler (Sørensen et al. 1996) that has call-by-name semantics in Table 3.1.

If the driving algorithm is extended according to the suggestions in Section 3.7.1 we add the possibility to evaluate certain expressions to their respective value. This is beyond what the rules for the positive (Sørensen et al. 1996) and perfect supercompiler (Secher 1999) does.

The question whether it is beneficial to replace an expression with its value remains unanswered throughout this work. In a system where evaluation is cheap and storage is expensive it might be beneficial to leave the functions instead of their values.

Table 3.1: Taxonomy of transformers, with our positive supercompiler

Transformer	Information Propagation	Eval. strategy	Data structure removal
Partial evaluation	Constant	CBV	No
Deforestation	Constant	CBN	Yes
Positive SCP, CBV	Unification	CBN	Yes
Positive SCP	Unification	CBN	Yes
Perfect SCP	Constraint	CBN	Yes
GPC	Constraint	CBN	Yes

3.8.3 Closest relative

The positive supercompiler (Sørensen et al. 1996) is the work which is closest to our work, especially if looking in Table 3.1. Many of the lessons learned in the perfect supercompiler (Secher 1999; Secher and Sørensen 2000) should be directly applicable if one chose to convert this positive supercompiler to a perfect one.

CHAPTER 4

Measurements

> Problems worthy of attack prove their worth by fighting back.
> – *Paul Erdos*

We provide measurements from a set of common examples from the litterature on deforestation. We show that our positive supercompiler does remove intermediate structures, and can improve the performance by an order of magnitude for certain benchmarks.

4.1 Experimental Setup

The measurements were performed on a Macbook Pro with a 2 GHz Intel Core Duo processor and 1 GB 667 MHz RAM running Mac OS X 10.4.11. Running the command *uname -a* reports Darwin Kernel Version 8.11.1 compiled Wed Oct 10 18:23:28 PDT 2007

GCC version 4.0.1 (Apple Computer, Inc. build 5367) as included by XCode 2.4.0 was used as compiler for the C output by the Timber compiler. The Timber compiler used was the most recent version available from darcs, with the last patch commited Fri Nov 9 16:33:29

The positive supercompiler was modified to leave all expressions bound to variables named *nodrive* as they were before transformation. This gives a possibility to construct the data necessary for the benchmark in equal time for both the optimized and un-optimized case.

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

4.1.1 Measuring time

Modern processors from Intel and AMD contain an instruction called RDTSC (Read Time Stamp Counter). This time stamp counter is simply the number of clock ticks since the last

```
-    putStr(ROOT(w));
+    before = rdtsc();
+    ret = (LIST)ROOT(w);
+    after = rdtsc();
+    printf("%llu\n", after-before);
+    putStr(ret);
```

Figure 4.1: Modifications to measure time

```
-#define NEW(t,lhs,sz) lhs = (t)hp; hp += WORDS(sz); \
-                      if (hp >= lim) lhs = (t)force(WORDS(sz));
+#define NEW(t,lhs,sz) totsize += sz; totcalls++; \
+                      lhs = (t)hp; hp += WORDS(sz); \
+                      if (hp >= lim) lhs = (t)force(WORDS(sz));
```

Figure 4.2: Modifications to measure allocations

power-on or hardware reset. We use code by Yoshii (2007) and modify the Timber runtime system to read the time stamp counter before and after executing the root function, as shown in Figure 4.1. Notice that the processing of command line arguments and the output of the result is not included in the time measured.

4.1.2 Measuring allocations

The runtime system contains a macro *NEW* that is used for all memory allocations. We modify that macro to increment two counters, one tracking the number of calls to *NEW* and the second tracking the number of bytes allocated in total. The patch is shown in Figure 4.2.

4.2 Results

The raw data for the time and size measurements are shown in Table 4.1. The time column is number of clockticks from the RDTSC instruction and the binary size is in bytes as reported by *ls*. The raw data for the memory allcation measurements are shown in Table 4.2. The total number of allocations and the total memory size allocated by the program are displayed in each column.

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

	Time		Binary size	
Benchmark	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 Square 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 4.1: Time and space measurements

	Allocations		Alloc Size	
Benchmark	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 Square of a Tree	4 194 338	91	29 360 496	908
Kort's Raytracer	60 021	17	320 144	124

Table 4.2: Allocation measurements

4.2.1 Double Append

As previously seen, appending three lists saves one traversal over the first list. This is an example by Wadler (1990), implemented in Timber as:

$$
\begin{aligned}
&append\ xs\ ys = \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\qquad [] \rightarrow ys\\
&\qquad\qquad (x' : xs') \rightarrow x' : (append\ xs'\ ys)\\
&root\ args \quad = append\ (append\ nodrive1\ nodrive2)\ nodrive3\\
&\quad \mathbf{where}\\
&\qquad num = str2int\ (getArg\ 1\ args)\\
&\qquad nodrive1 = concat\ (replicate\ num\ (getArg\ 2))\\
&\qquad nodrive2 = concat\ (replicate\ num\ (getArg\ 3))\\
&\qquad nodrive3 = concat\ (replicate\ num\ (getArg\ 4))
\end{aligned}
$$

The implemented driving algorithm gives the expected result:

$$
\begin{aligned}
&root\ args \quad = h1\ nodrive1\ nodrive2\ nodrive3\\
&\quad \mathbf{where}\\
&\qquad num = str2int\ (getArg\ 1\ args)\\
&\qquad nodrive1 = concat\ (replicate\ num\ (getArg\ 2))\\
&\qquad nodrive2 = concat\ (replicate\ num\ (getArg\ 3))\\
&\qquad nodrive3 = concat\ (replicate\ num\ (getArg\ 4))
\end{aligned}
$$

$$
\begin{aligned}
h1\ xs_1\ ys_1\ zs_1 = \mathbf{case}\ xs_1\ \mathbf{of}& \\
&[]\ \rightarrow\ \mathbf{case}\ ys_1\ \mathbf{of} \\
&\quad []\ \rightarrow\ zs_1 \\
&\quad (y_1' : ys_1')\ \rightarrow\ y_1' : (h2\ ys_1'\ zs_1) \\
&\quad \_\ \rightarrow\ raise\ 1 \\
&(x_1' : xs_1')\ \rightarrow\ x_1' : (h1\ xs_1'\ ys_1\ zs_1) \\
&\_\ \rightarrow\ raise\ 1 \\
h2\ xs_2\ ys_2 \quad = \mathbf{case}\ xs_2\ \mathbf{of}& \\
&[]\ \rightarrow\ ys_2 \\
&(x_2' : xs_2')\ \rightarrow\ x_2' : (h2\ xs_2'\ ys_2) \\
&\_\ \rightarrow\ raise\ 1
\end{aligned}
$$

The arguments for the benchmark was "3000 foo bar baz". This builds three strings of 9000 characters each, that are appended to eachother into a 27 000 characters long string. The number of allocations goes down, and one iteration over the first string is avoided. The binary size increase 1316 bytes, on a binary of roughly 90k.

4.2.2 Factorial

Factorial is the functional programming equivalent of "Hello world" in the imperative world. There are no intermediate lists created in a standard implementation, so any performance improvements will come from inlining or reductions.

$$
\begin{aligned}
fac\ 0 &= 1 \\
fac\ n &= n * fac\ (n-1)
\end{aligned}
$$

$$root\ xs = show\ (fac\ 3)$$

This is transformed to:

$$root\ xs = show\ (3 * fac\ 2)$$

This saves one recursion and a couple of reductions. The measurements confirm that the allocations remain the same, but the runtime is slightly reduced. The final binary size remains unchanged.

4.2.3 Flip a Tree

Flipping a tree is another example by Wadler (1990), and in this case we perform a double flip (thus restoring the original tree) before printing the total sum of all leaves. 12 was given as argument for the measurements.

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

$$
\begin{aligned}
buildTree\ 0\ t\ &=\ t \\
buildTree\ n\ t\ &=\ buildTree\ (n-1)\ (Branch\ t\ t)
\end{aligned}
$$

```
sumtr (Leaf a) = a
sumtr (Branch l r) = sumtr l + sumtr r

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

root args = let ys = (flip (flip nodrive1)) in show (sumtr ys)
  where
    num = (str2int (getArg 1 args))
    nodrive1 = buildTree num (Leaf 3)
```

This is transformed into:

```
root args = show ( case nodrive1 of
                     Leaf d → d
                     Branch l r → (h1 l) + (h1 r)
                     _ → raise 1 )
  where
    nodrive1 = buildTree (str2int (getArg 1 args)) (Leaf 3)
h1 t = case t of
         Leaf d → d
         Branch l r → (h1 l) + (h1 r)
         _ → raise 1
```

The function $h1$ is isomorphic to *sumtr* in the input program, and the double flip has been eliminated. Both total number of allocations and the total size of allocations is reduced. The runtime also goes down by an order of magnitude! The binary size increases close to 10% however.

4.2.4 Sum of Square of a Tree

Computing the sum of the square of a tree is the final example by Wadler (1990). The measurements were done with the argument 20.

```
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)
```

```
buildTree 0 t = t
buildTree n t = buildTree (n - 1) (Branch t t)

root args = show (sumtr (squaretr nodrive1))
  where
    num = str2int (getArg 1 args)
    nodrive1 = buildTree num (Leaf 3)
```

This is transformed to:

```
root args = show ( case nodrive1 of
                     Leaf d -> d * d
                     Branch l r -> (h1 l) + (h1 r)
                     _ -> raise 1)
  where
    nodrive1 = buildTree (str2int (getArg 1 args)) (Leaf 3))
h1 t = case t of
         Leaf d -> d * d
         Branch l r -> (h1 l) + (h1 r)
         _ -> raise 1
```

Almost all allocations are removed, but the binary size is increased by nearly 10% in this example too. The runtime is however improved by an order of magnitude in this example too.

4.2.5 Kort's Raytracer

The inner loop of a Raytracer (Kort 1996) is extracted and transformed. 20000 is the argument to the program.

```
zipWith f (h1 : t1) (h2 : t2) = (f h1 h2) : zipWith f t1 t2
zipWith _ _ _ = []

sum :: [Int] -> Int
sum [] = 0
sum (h : t) = h + sum t

root args = show (sum (zipWith (*) nodrive1 nodrive2))
  where
    num = str2int (getArg 1 args)
    nodrive1 :: [Int]
    nodrive1 = replicate num 3
    nodrive2 :: [Int]
    nodrive2 = replicate num 5
```

The transformed result:

```
root args = show (h1 nodrive1 nodrive2)
  where
    nodrive1 :: [Int]
    nodrive1 = replicate (str2int (getArg 1 args)) 3
    nodrive2 :: [Int]
    nodrive2 = replicate (str2int (getArg 1 args)) 5
h1 xs ys = case xs of
             (x' : xs') -> case ys of
                             (y' : ys') -> (x' * y') + (h1 xs' ys')
                             _ -> 0
             _ -> 0
```

The total runtime, the number of allocations, the total size of allocations and the binary size are all decreased.

CHAPTER 5

Conclusions and Future Research

A witty saying proves nothing.
– *Voltaire*

5.1 Conclusions

A positive supercompiler, for a higher-order call-by-value language, that includes folding has been presented. It was proven correct and its relation to classic compiler optimizations was investigated.

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

5.2 Summary of Contributions

- The most important piece of this work is the positive supercompilation algorithm for a higher-order call-by-value language, including folding, outlined in Chapter 3.
- The same chapter contains proofs for preservation of semantics of the positive supercompiler.
- The algorithm is proven to terminate for all inputs.
- We show how to ensure that transformed programs are at least as efficient as the original program.
- Several extensions to the positive supercompiler are discussed, giving it the possibility to evaluate closed expressions and replace them with their respective value instead.

5.3 Future Research

This thesis has covered the first semantic layer of Timber (Carlsson et al. 2003). Merging objects on the second layer is a candidate for future work and Liu et al. (2005) would be a good starting point for this work.

The improved driving of applications in Section 3.6 is the first thing that needs to be investigated. Proving the total correctness and termination of the algorithm with this extension is necessary. Having a mechanism to split expressions will also open up for handling primitive operations in a better way.

Forthcoming work will investigate how the concept of an inline budget may be used to obtain good balance between code size and inlining benefits. Such a budget needs to be tuned as well, so a calculus for controlling inlining is necessary. This budget could possibly make deforestation (Marlow 1995) a more attractive candidate for GHC as well.

Supercompiling objects together might change schedulability properties for the program as whole, at least in a system with several processors, since each object runs in its own thread. Consider the case of two processors and two tasks, each task requiring $(50 + \epsilon)\%$ process time per period. If merged, it would be scheduled on one processor and require $(100 + 2\epsilon)\%$ process time per period, from one processor. Clearly this is not possible, whereas the original program was schedulable and had $(50 - \epsilon)\%$ slack per processor each period.

It is obvious that more work could be done on the components of our supercompiler, mainly strictness and termination analysis. It is not however our intention to follow that track, but rather use the results of others.

The perfect supercompiler propagates both positive and negative information, it would be interesting to compare that with the full strength obtained by the theorem prover in GPC as documented by Futamura et al. (2002).

APPENDIX A

Proofs

> On theories such as these we cannot rely.
> Proof we need. Proof!
> – *Yoda*

The complete proof for total correctness of the driving algorithm presented in Chapter 3 is presented.

A.1 Proof of Lemma 3.13

For each rule R: $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho} = e_1$ in definition 3.5 and 3.7 and each recursive application $\mathcal{D}[\![e']\!]_{\mathcal{R}',\rho'}$ in e_1, $|\mathcal{D}[\![\ e']\!]_{\mathcal{R}',\rho'}| < |\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}|$

Proof. Inspection of rules:

R1 We have LHS $\mathcal{D}[\![n]\!]_{\mathcal{R},\mathcal{G},\rho}$ and no recursive applications in the RHS, so Lemma 3.13 holds vacuously.

R2 We have LHS $\mathcal{D}[\![x]\!]_{\mathcal{R},\mathcal{G},\rho}$ and no recursive applications in the RHS, so Lemma 3.13 holds vacuously.

R3 Argument analogous to R9.

R4 We have LHS $\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho}$ and recursive applications $\mathcal{D}[\![e']\!]_{[],\mathcal{G},\rho}$ for each *e'* in $\overline{e}$. $|\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho}|$ $= (N - |\rho|, |k\,\overline{e}|, |k\,\overline{e}|)$ and $|\mathcal{D}[\![e']\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e'|, |e'|)$. From Definition 3.4 it follows that $|e'| < |k\,\overline{e}|$ and by Definition 3.12 $(N - |\rho|, |e'|, |e'|) < (N - |\rho|, |k\,\overline{e}|, |k\,\overline{e}|)$ for all *e'* in $\overline{e}$

R5 We have LHS $\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive applications $\mathcal{D}[\![e']\!]_{[],\mathcal{G},\rho}$ for each *e'* in $\overline{e}$. $|\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}|$ $= (N - |\rho|, |\mathcal{R}\langle x\,\overline{e}\rangle|, |x\,\overline{e}|)$ and $|\mathcal{D}[\![e']\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e'|, |e'|)$. From Definition 3.4 it follows that $|e'| < |\mathcal{R}\langle x\,\overline{e}\rangle|$ and by Definition 3.12 $(N - |\rho|, |e'|, |e'|) < (N - |\rho|, |\mathcal{R}\langle x\,\overline{e}\rangle|, |x\,\overline{e}|)$ for all *e'* in $\overline{e}$

R6 We have LHS $\mathcal{D}[\![\lambda\,\overline{x}.e]\!]_{[],\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\lambda\overline{x}.e|, |\lambda\overline{x}.e|)$ and $|\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e|, |e|)$. From Definition 3.4 it follows that $|e| < |\lambda\overline{x}.e|$ and by Definition 3.12 $(N - |\rho|, |e|, |e|) < (N - |\rho|, |\lambda\overline{x}.e|, |\lambda\overline{x}.e|)$.

R7 We have LHS $\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho}|$ $= (N - |\rho|, |\mathcal{R}\langle n_1 \oplus n_2\rangle|, |n_1 \oplus n_2|)$ and $|\mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle n\rangle|, |\mathcal{R}\langle n\rangle|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle n\rangle| < |\mathcal{R}\langle n_1 \oplus n_2\rangle|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle n\rangle|,$ $|\mathcal{R}\langle n\rangle|) < (N - |\rho|, |\mathcal{R}\langle n_1 \oplus n_2\rangle|, |n_1 \oplus n_2|)$.

R8 We have LHS $\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive applications $\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho}$ and $\mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle e_1 \oplus e_2\rangle|, |e_1 \oplus e_2|)$, $|\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e_1|, |e_1|)$ and $|\mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e_2|, |e_2|)$. From Definition 3.4 it follows that $|e_1| < |\mathcal{R}\langle e_1 \oplus e_2\rangle|$, $|e_2| < |\mathcal{R}\langle e_1 \oplus e_2\rangle|$ and by Definition 3.12 $(N - |\rho|, |e_1|, |e_1|) < (N - |\rho|, |\mathcal{R}\langle e_1 \oplus e_2\rangle|, |e_1 \oplus e_2|)$ and $(N - |\rho|, |e_2|, |e_2|) < (N - |\rho|, |\mathcal{R}\langle e_1 \oplus e_2\rangle|, |e_1 \oplus e_2|)$.

R9 a) We have LHS $\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and no recursive applications on the RHS, so Lemma 3.13 holds vacuously.

b) We have LHS $\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive applications $\mathcal{D}[\![e']\!]_{[],\mathcal{G},\rho}$ for each *e'* in $\overline{e}$. $|\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle g\,\overline{e}\rangle|, |g\,\overline{e}|)$ and $|\mathcal{D}[\![e']\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e'|, |e'|)$. From Definition 3.4 it follows that $|e'| < |\mathcal{R}\langle g\,\overline{e}\rangle|$ and by definition 3.12 $(N - |\rho|, |e'|, |e'|) <$ $(N - |\rho|, |\mathcal{R}\langle g\,\overline{e}\rangle|, |g\,\overline{e}|)$ for all *e'* in $\overline{e}$

c) We have LHS $\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}$. $|\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}| =$ $(N - |\rho|, |\mathcal{R}\langle g\,\overline{e}\rangle|, |g\,\overline{e}|)$ and $|\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}| = (N - |\rho'|, |\mathcal{R}\langle v\,\overline{e}\rangle|, |\mathcal{R}\langle v\,\overline{e}\rangle|)$. The length of ρ' is one element longer than the length of ρ and by definition 3.12 $(N -$ $|\rho'|, |\mathcal{R}\langle v\,\overline{e}\rangle|, |\mathcal{R}\langle v\,\overline{e}\rangle|) < (N - |\rho|, |\mathcal{R}\langle g\,\overline{e}\rangle|, |g\,\overline{e}|)$

R10 We have LHS $\mathcal{D}[\![(\lambda\,\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}$. $|\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle(\lambda\overline{x}.f)\,\overline{e}\rangle|, |(\lambda\overline{x}.f)\,\overline{e}|)$ and $|\mathcal{D}[\![\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho} = (N -$ $|\rho|, |\mathcal{R}\langle\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f\rangle|, |\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f|)$. From Definition 3.4 it follows that $|\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f| <$ $|(\lambda\overline{x}.f)\,\overline{e}|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f\rangle|, |\textbf{let}\,\overline{x} = \overline{e}\,\textbf{in}\,f|) < (N -$ $|\rho|, |\mathcal{R}\langle(\lambda\overline{x}.f)\,\overline{e}\rangle|, |(\lambda\overline{x}.f)\,\overline{e}|)$.

R11 We have LHS $\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,\overline{e}\rangle,\mathcal{G},\rho}$. $|\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N -$ $|\rho|, |\mathcal{R}\langle e\,\overline{e}\rangle|, |e\,\overline{e}|)$ and $|\mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,\overline{e}\rangle,\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle e\,\overline{e}\rangle|, |e|)$. From Definition 3.4 it follows that $|e| < |e\,\overline{e}|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle e\,\overline{e}\rangle|, |e|) < (N - |\rho|, |\mathcal{R}\langle e\,\overline{e}\rangle|, |e\,\overline{e}|)$.

R12 We have LHS $\mathcal{D}[\![\textbf{let}\,x = v\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{let}\,x = v\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle\textbf{let}\,x = v\,\textbf{in}\,f\rangle|, |\textbf{let}\,x = v\,\textbf{in}\,f|)$ and $|\mathcal{D}[\![\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle[v/x]f\rangle|, |\mathcal{R}\langle[v/x]f\rangle|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle[v/x]f\rangle| < |\mathcal{R}\langle\textbf{let}\,x = v\,\textbf{in}\,f\rangle|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle[v/x]f\rangle|, |\mathcal{R}\langle[v/x]f\rangle|)$ $< (N - |\rho|, |\mathcal{R}\langle\textbf{let}\,x = v\,\textbf{in}\,f\rangle|, |\textbf{let}\,x = v\,\textbf{in}\,f|)$.

R13 We have LHS $\mathcal{D}[\![\textbf{let}\,x = y\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{let}\,x = y\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle\textbf{let}\,x = y\,\textbf{in}\,f\rangle|, |\textbf{let}\,x = y\,\textbf{in}\,f|)$ and $|\mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle[y/x]f\rangle|, |\mathcal{R}\langle[y/x]f\rangle|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle[y/x]f\rangle| < |\mathcal{R}\langle\textbf{let}\,x = y\,\textbf{in}\,f\rangle|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle[y/x]f\rangle|, |\mathcal{R}\langle[y/x]f\rangle|)$ $< (N - |\rho|, |\mathcal{R}\langle\textbf{let}\,x = y\,\textbf{in}\,f\rangle|, |\textbf{let}\,x = y\,\textbf{in}\,f|)$.

R14 a) We have LHS $\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|, |\textbf{let } x = e \textbf{ in } f|)$ and $|\mathcal{D}[\![\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle [e/x]f\rangle|, |\mathcal{R}\langle [e/x]f\rangle|)$. Since f is linear with respect to x and from Definition 3.4 it follows that $|\mathcal{R}\langle [e/x]f\rangle| < |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle [e/x]f\rangle|, |\mathcal{R}\langle [e/x]f\rangle|) < (N - |\rho|,$ $|\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|, |\textbf{let } x = e \textbf{ in } f|)$.

b) We have LHS $\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive applications $\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}$ and $\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|, |\textbf{let } x = e \textbf{ in } f|)$, $|\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle f\rangle|, |\mathcal{R}\langle f\rangle|)$ and $|\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |e|, |e|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle f\rangle| < |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|$ and $|e| < |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|$ and by Definition 3.12 $(N-|\rho|, |\mathcal{R}\langle f\rangle|, |\mathcal{R}\langle f\rangle|) < (N-|\rho|, |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|, |\textbf{let } x = e \textbf{ in } f|)$ and $(N - |\rho|, |e|, |e|) < (N - |\rho|, |\mathcal{R}\langle \textbf{let } x = e \textbf{ in } f\rangle|, |\textbf{let } x = e \textbf{ in } f|)$

R15 We have LHS $\mathcal{D}[\![\textbf{letrec } g = v \textbf{ in } e]\!]_{\mathcal{R},\mathcal{G},\rho}$ and the recursive application $\mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{letrec } g = v \textbf{ in } e]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle \textbf{letrec } g = v \textbf{ in } e\rangle|, |\textbf{letrec } g = v \textbf{ in } e|)$, $|\mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle e\rangle|, |\mathcal{R}\langle e\rangle|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle e\rangle| < |\mathcal{R}\langle \textbf{letrec } g = v \textbf{ in } e\rangle|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle e\rangle|, |\mathcal{R}\langle e\rangle|) < (N - |\rho|,$ $|\mathcal{R}\langle \textbf{letrec } g = v \textbf{ in } e\rangle|, |\textbf{letrec } g = v \textbf{ in } e|)$.

R16 We have LHS $\mathcal{D}[\![\textbf{case } x \textbf{ of } \{ p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive applications $\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}$ for each e_i. $|\mathcal{D}[\![\textbf{case } x \textbf{ of } \{p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N-|\rho|, |\mathcal{R}\langle \textbf{case } x \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } x \textbf{ of } \{p_i \rightarrow e_i\}|)$ and $|\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle e_i\rangle|, |\mathcal{R}\langle e_i\rangle|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle e_i\rangle| < |\mathcal{R}\langle \textbf{case } x \textbf{ of } \{p_i \rightarrow e_i\}\rangle|$ and by definition 3.12 $(N - |\rho|, |\mathcal{R}\langle e_i\rangle|, |\mathcal{R}\langle e_i\rangle|) < (N - |\rho|, |\mathcal{R}\langle \textbf{case } x \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } x \textbf{ of } \{p_i \rightarrow e_i\}|)$ for all e_i.

R17 We have LHS $\mathcal{D}[\![\textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and the recursive application $\mathcal{D}[\![\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j]\!]_{\mathcal{R},\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|,$ $|\mathcal{R}\langle \textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}|)$ and $|\mathcal{D}[\![\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle \textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j\rangle|, |\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j|)$. From Definition 3.4 it follows that $|\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j| < |\textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}|$ and by Definition 3.12 $(N-|\rho|, |\mathcal{R}\langle \textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j\rangle|, |\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j|) < (N - |\rho|,$ $|\mathcal{R}\langle \textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } k_j\, \overline{e} \textbf{ of } \{p_i \rightarrow e_i\}|)$.

R18 We have LHS $\mathcal{D}[\![\textbf{case } n_j \textbf{ of } \{ p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{case } n_j \textbf{ of } \{p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N - |\rho|, |\mathcal{R}\langle \textbf{case } n_j \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } n_j \textbf{ of } \{p_i \rightarrow e_i\}|)$ and $|\mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho}| = (N-|\rho|, |\mathcal{R}\langle e_j\rangle|, |\mathcal{R}\langle e_j\rangle|)$. From Definition 3.4 it follows that $|\mathcal{R}\langle e_j\rangle| < |\mathcal{R}\langle \textbf{case } n_j \textbf{ of } \{p_i \rightarrow e_i\}\rangle|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle e_j\rangle|, |\mathcal{R}\langle e_j\rangle|) < (N - |\rho|,$ $|\mathcal{R}\langle \textbf{case } n_j \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } n_j \textbf{ of } \{p_i \rightarrow e_i\}|)$.

R19 Argument analogous to R16.

R20 We have LHS $\mathcal{D}[\![\textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}$ and recursive application $\mathcal{D}[\![e]\!]_{\mathcal{R}\langle \textbf{case } [] \textbf{ of } \{p_i \rightarrow e_i\}\rangle,\mathcal{G},\rho}$. $|\mathcal{D}[\![\textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}| = (N-|\rho|, |\mathcal{R}\langle \textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}\rangle|,$ $|\textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}|)$ and $|\mathcal{D}[\![e]\!]_{\mathcal{R}\langle \textbf{case } [] \textbf{ of } \{p_i \rightarrow e_i\}\rangle,\mathcal{G},\rho}| = (N-|\rho|, |\mathcal{R}\langle \textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |e|)$. From Definition 3.4 it follows that $|e| < |\textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}|$ and by Definition 3.12 $(N - |\rho|, |\mathcal{R}\langle \textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |e|) < (N - |\rho|, |\mathcal{R}\langle \textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}\rangle|, |\textbf{case } e \textbf{ of } \{p_i \rightarrow e_i\}|)$.

□

A.2 Proof of Lemma 3.14

For all well-typed expressions e, $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}$ is matched by a unique rule in Figure 3.5.

Proof. Case analysis over all terms for $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}$:

$e = n$ Rule R1 matches.

$e = x$ Rule R2 matches.

$e = g$ Rules R3/R9 match.

$e = f\,\overline{e}$ Subcases:

- Case $f = n$: Type error.
- Case $f = x$: Rule R5 matches.
- Case $f = g$: Rule R9 matches.
- Case $f = f'\,\overline{e'}$: Rule R11 matches.
- Case $f = \lambda\overline{x}.e'$: Rule R10 matches.
- Case $f = k\,\overline{e'}$: Type error.
- Case $f = e_1 \oplus e_2$: Type error.
- Case $f = \textbf{let}\ x = e\ \textbf{in}\ f'$: Rule R11 matches.
- Case $f = \textbf{letrec}\ g = v\ \textbf{in}\ e$: Rule R11 matches.
- Case $f = \textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\}$: Rule R11 matches.

$e = \lambda\overline{x}.e$ Rule R6 matches.

$e = k\,\overline{e}$ Rule R4 matches.

$e = e_1 \oplus e_2$ Rule R7/R8 matches.

$e = \textbf{let}\ x = e\ \textbf{in}\ f$ Rule R12/R13/R14 matches.

$e = \textbf{letrec}\ g = v\ \textbf{in}\ f$ Rule R15 matches.

$e = \textbf{case}\ f\ \textbf{of}\ \{p_i \to e_i\}$ Subcases:

- Case $f = n$: Rule R18 matches.
- Case $f = x$: Rule R16 matches.
- Case $f = g$: Rule R20 matches

- Case $f = f'\,\overline{e'}$:
  - Case $f' = x$: Rule R19 matches.
  - Otherwise: Rule R20 matches.
- Case $f = \lambda\overline{x}.e'$: Type error.
- Case $f = k\,\overline{e}$: Rule R17 matches.
- Case $f = e_1 \oplus e_2$: Rule R19 matches.
- Case $f = \textbf{let}\ x = e\ \textbf{in}\ f'$: Rule R20 matches.
- Case $f = \textbf{letrec}\ g = v\ \textbf{in}\ f'$: Rule R20 matches.
- Case $f = \textbf{case}\ e\ \textbf{of}\ \{p_i \to e_i\}$: Rule R20 matches.

□

A.3 Proof of Supporting Lemmas

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

Lemma A.1 (Sands, p. 24)**.** *For all expressions e and value substitutions 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 $\overline{x} \notin (fv(h) \cup fv(\lambda n.fix\,(\lambda h.\lambda\overline{x}.e_1)\,n)$:

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

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 3.26 the initial expressions are cost equivalent. □

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

Proof (Similar to Sands (1996a)). By inspection of the rules for $\mathcal{D}[\![\,]\!]$, all free occurences of *h* in $\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}$ must occur in sub-expressions of the form $h\,\overline{x}$. Suppose there are *k* such occurences, 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 $\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}$ 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 occurences of *h*. Then (substitution associates to the right):

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

□

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

Proof. Notice that $\mathcal{R}\langle\textbf{let}\,x = []\,\textbf{in}\,f\rangle$ is a redex, and assume $e \mapsto^k v$. The LHS evaluates in k steps $\mathcal{R}\langle\textbf{let}\,x = e\,\textbf{in}\,f\rangle \mapsto^k \mathcal{R}\langle\textbf{let}\,x = v\,\textbf{in}\,f\rangle \mapsto \mathcal{R}\langle[v/x]f\rangle$, and the RHS evaluates in k steps $\textbf{let}\,x = e\,\textbf{in}\,\mathcal{R}\langle f\rangle \mapsto^k \textbf{let}\,x = v\,\textbf{in}\,\mathcal{R}\langle f\rangle \mapsto [v/x]\mathcal{R}\langle f\rangle$. Since contexts do not bind variables these two terms are equivalent and by Lemma 3.26 the initial terms are cost equivalent. □

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

Proof. Translate both sides according to 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 $\mathcal{R}\langle[]\rangle$ is a redex. The LHS evaluates in 0 steps $\mathcal{R}\langle(\lambda g.e)\,(\lambda n.fix\,(\lambda g.v)\,n)\rangle \mapsto \mathcal{R}\langle[\lambda n.fix\,(\lambda g.v)\,n/g]e\rangle$ and the RHS $(\lambda g.\mathcal{R}\langle e\rangle)\,(\lambda n.fix\,(\lambda g.v)\,n) \mapsto [\lambda n.fix\,(\lambda g.v)\,n/g]\mathcal{R}\langle e\rangle$. Since our contexts do not bind variables these two terms are equivalent and by Lemma 3.26 the initial terms are cost equivalent. □

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

Proof. Notice that $\mathcal{R}\langle \textbf{case}\, []\, \textbf{of}\, \{p_i \to e_i\}\rangle$ is a redex, and assume $e \mapsto^k n_j$. The LHS evaluates in k steps $\mathcal{R}\langle \textbf{case}\, e\, \textbf{of}\, \{p_i \to e_i\}\rangle \mapsto^k \mathcal{R}\langle \textbf{case}\, n_j\, \textbf{of}\, \{p_i \to e_i\}\rangle \mapsto \mathcal{R}\langle e_j\rangle$, and the RHS evaluates in k steps $\textbf{case}\, e\, \textbf{of}\, \{p_i \to \mathcal{R}\langle e_i\rangle\} \mapsto^k \textbf{case}\, n_j\, \textbf{of}\, \{p_i \to \mathcal{R}\langle e_i\rangle\}\mathcal{R}\langle f\rangle \mapsto \mathcal{R}\langle e_j\rangle$. Since these two terms are equivalent the initial terms are cost equivalent by Lemma 3.26.

□

A.4 Proof of Proposition 3.32

We set out to prove the proposition that if e is an expression, and ρ an environment such that

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

then $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})$. We reason by induction on the structure of expressions, and since the algorithm is total (Lemma 3.14) this coincides with inspection of each rule.

A.4.1 R1

We have that $\rho(\mathcal{D}[\![n]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{R}\langle n\rangle)$, 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 $\mathcal{R}\langle n\rangle \trianglerighteq_s \rho(\mathcal{D}[\![n]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.2 R2

We have that $\rho(\mathcal{D}[\![x]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{R}\langle x\rangle)$, 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 $\mathcal{R}\langle x\rangle \trianglerighteq_s \rho(\mathcal{D}[\![x]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.3 R3

Argument analogous to R9.

A.4.4 R4

We have that $\rho(\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho}) = \rho(k\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})$, and the conditions of the proposition ensure that $fv(k\,\overline{e}) \cap dom(\rho) = \emptyset$, so $\rho(k\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}) = k\,\rho(\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis, $\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})$, and by congruence properties of strong improvement (Lemma 3.28:1) $k\,\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho})$.

A.4.5 R5

We have that $\rho(\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle x\,\overline{e}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle) = \mathcal{R}\langle x\,\rho(\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})\rangle$. By the induction hypothesis, $\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})$, and by congruence properties of strong improvement (Lemma 3.28:1) $\mathcal{R}\langle x\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.6 R6

We have that $\rho(\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{G},\rho}) = \rho(\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})$, and the conditions of the proposition ensure that $fv(\lambda\overline{x}.e) \cap dom(\rho) = \emptyset$, so $\rho(\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}) = \lambda\overline{x}.\rho(\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis, $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})$, and by congruence properties of strong improvement (Lemma 3.28:1) $\lambda\overline{x}.e \trianglerighteq_s \rho(\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{G},\rho})$.

A.4.7 R7

We have that $\rho(\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis, $\mathcal{R}\langle n\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho})$, and since $\mathcal{R}\langle n_1 \oplus n_2\rangle \mapsto \mathcal{R}\langle n\rangle$ it follows from Lemma 3.28:4 that $\mathcal{R}\langle n_1 \oplus n_2\rangle \trianglerighteq_s \rho(\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.8 R8

We have that $\rho(\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{R}\langle\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho} \oplus \mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle e_1 \oplus e_2\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho} \oplus \mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}\rangle) = \mathcal{R}\langle\rho(\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho}) \oplus \rho(\mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho})\rangle$. By the induction hypothesis $e_1 \trianglerighteq_s \rho(\mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho})$ and $e_2 \trianglerighteq_s \rho(\mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho})$, and by congruence properties of strong improvement (Lemma 3.28:1) $\mathcal{R}\langle e_1 \oplus e_2\rangle \trianglerighteq_s \rho(\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.9 R9

A.4.9.1 Case: found in memolist

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

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

A.4.9.2 Case: Whistle blows

We have that $\rho(\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{R}\langle g\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle)$, and the conditions of the proposition ensure that $fv(\mathcal{R}\langle g\,\overline{e}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathcal{R}\langle g\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle) = \mathcal{R}\langle g\,\rho(\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})\rangle$. By the induction hypothesis, $\overline{e} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho})$, and by congruence properties of strong improvement (Lemma 3.28:1) $\mathcal{R}\langle g\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.9.3 Case: Add to memolist

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

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

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

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

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

By Theorem 3.31 it is sufficient to show:

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

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

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

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

A.4.9.4 Case: Inline

We have that $\rho(\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis $\mathcal{R}\langle v\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle v\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho})$, and since $\mathcal{R}\langle g\,\overline{e}\rangle \mapsto^1 \mathcal{R}\langle v\,\overline{e}\rangle$ it follows from Lemma 3.28:4 that $\mathcal{R}\langle g\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.10 R10

We have that $\rho(\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\,\overline{x} = \overline{e}\,\mathbf{in}\,f\rangle]\!]_{[],\mathcal{G},\rho})$. 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 3.26 their ancestor terms

are cost equivalent, $\mathcal{R}\langle(\lambda\overline{x}.f)\,\overline{e}\rangle \trianglelefteq\!\trianglerighteq \mathcal{R}\langle\mathbf{let}\,\overline{x}=\overline{e}\,\mathbf{in}\,f\rangle$, and cost equivalence implies strong improvement. By the induction hypothesis $\mathcal{R}\langle\mathbf{let}\,\overline{x}=\overline{e}\,\mathbf{in}\,f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\,\overline{x}=\overline{e}\,\mathbf{in}\,f\rangle]\!]_{[],\mathcal{G},\rho})$, and therefore $\mathcal{R}\langle(\lambda\overline{x}.f)\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.11 R11

We have that $\rho(\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,\overline{e}\rangle,\mathcal{G},\rho})$ and $\mathcal{R}\langle e\,\overline{e}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,\overline{e}\rangle,\mathcal{G},\rho})$ follows from the induction hypothesis.

A.4.12 R12

We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x=v\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis $\mathcal{R}\langle[v/x]f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\mathcal{G},\rho})$, and since $\mathcal{R}\langle\mathbf{let}\,x=v\,\mathbf{in}\,f\rangle \mapsto \mathcal{R}\langle[v/x]f\rangle$ it follows from Lemma 3.28:4 that $\mathcal{R}\langle\mathbf{let}\,x=v\,\mathbf{in}\,f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,x=v\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.13 R13

We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x=y\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis $\mathcal{R}\langle[y/x]f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\mathcal{G},\rho})$, and since $\mathcal{R}\langle\mathbf{let}\,x=y\,\mathbf{in}\,f\rangle \trianglelefteq\!\trianglerighteq \mathcal{R}\langle[y/x]f\rangle$ it follows that $\mathcal{R}\langle\mathbf{let}\,x=y\,\mathbf{in}\,f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,x=y\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.14 R14

A.4.14.1 Case: $x \in strict(f)$

We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x=e\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{[],\mathcal{G},\rho})$. Evaluating the input term 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 3.26 their ancestor terms are cost equivalent, $\mathcal{R}\langle\mathbf{let}\,x=e\,\mathbf{in}\,f\rangle \trianglelefteq\!\trianglerighteq \mathcal{R}\langle[e/x]f\rangle$, and cost equivalence implies strong improvement. By the induction hypothesis $\mathcal{R}\langle[e/x]f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{[],\mathcal{G},\rho})$, and therefore $\mathcal{R}\langle\mathbf{let}\,x=e\,\mathbf{in}\,f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,x=e\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.14.2 Case: otherwise

We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x=e\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathbf{let}\,x=\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}\,\mathbf{in}\,\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho})$, 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 $\rho(\mathbf{let}\,x = \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}\,\mathbf{in}\,\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}) = \mathbf{let}\,x = \rho(\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})\,\mathbf{in}\,\rho(\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho})$ and $\mathcal{R}\langle f\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho})$. By Lemma A.3 the input is strongly improved by $\mathbf{let}\,x=e\,\mathbf{in}\,\mathcal{R}\langle f\rangle$, and therefore $\mathcal{R}\langle\mathbf{let}\,x=e\,\mathbf{in}\,f\rangle \trianglerighteq_s$
$\rho(\mathcal{D}[\![\mathbf{let}\,x=e\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.15 R15

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

A.4.16 R16

We have that $\rho(\mathcal{D}[\![\mathbf{case}\, x\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathbf{case}\, x\, \mathbf{of}\, \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\})$, and the conditions of the proposition ensure that $\mathit{fv}(\mathcal{R}\langle\mathbf{case}\, x\, \mathbf{of}\, \{p_i \to e_i\}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathbf{case}\, x\, \mathbf{of}\, \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\}) = \mathbf{case}\, x\, \mathbf{of}\, \{p_i \to \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho})\}$. By the induction hypothesis $\mathcal{R}\langle e_i\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho})$. By 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 $\mathcal{R}\langle\mathbf{case}\, x\, \mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, x\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.17 R17

We have that $\rho(\mathcal{D}[\![\mathbf{case}\, k_j\, \overline{e}\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e}\, \mathbf{in}\, e_j\rangle]\!]_{[],\mathcal{G},\rho})$. 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 3.26 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. By the induction hypothesis $\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e}\, \mathbf{in}\, e_j\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle\mathbf{let}\, \overline{x}_j = \overline{e}\, \mathbf{in}\, e_j\rangle]\!]_{[],\mathcal{G},\rho})$, and therefore $\mathcal{R}\langle\mathbf{case}\, k_j\, \overline{e}\, \mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, k_j\, \overline{e}\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.18 R18

We have that $\rho(\mathcal{D}[\![\mathbf{case}\, n_j\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{),\mathcal{G},\rho} = \rho(\mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho})$. By the induction hypothesis $\mathcal{R}\langle e_j\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho})$, 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 3.28:4 that $\mathcal{R}\langle\mathbf{case}\, n_j\, \mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, n_j\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.19 R19

We have that $\rho(\mathcal{D}[\![\mathbf{case}\, a\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho}) = \rho(\mathbf{case}\, \mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho}\, \mathbf{of}\, \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\})$, and the conditions of the proposition ensure that $\mathit{fv}(\mathcal{R}\langle\mathbf{case}\, a\, \mathbf{of}\, \{p_i \to e_i\}\rangle) \cap dom(\rho) = \emptyset$, so $\rho(\mathbf{case}\, \mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho}\, \mathbf{of}\, \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\}) = \mathbf{case}\, \rho(\mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho})\, \mathbf{of}\, \{p_i \to \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho})\}$. By the induction hypothesis $a \trianglerighteq_s \rho(\mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho})$ and $\mathcal{R}\langle e_i\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho})$. 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 $\mathcal{R}\langle\mathbf{case}\, a\, \mathbf{of}\, \{p_i \to e_i\}\rangle \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{case}\, a\, \mathbf{of}\, \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho})$.

A.4.20 R20

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

REFERENCES

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

M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of Timber. *Lecture Notes in Computer Science*, 2895:339–356, January 2003.

A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones, and R. L. While. Exploring the barrier to entry: incremental generational garbage collection for haskell. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 163–174, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-945-4.

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

L. Damas and R. Milner. Principal type-schemes for functional prograias. In Richard DeMillo, editor, *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, NM, January 1982. ACM Press. ISBN 0-89791-065-6.

O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-388-X.

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

B. Fulgham. The great language shootout, January 2007. URL `http://shootout.alioth.debian.org/`.

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

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

GHC 2008. Glasgow Haskell Compiler, May 2008. URL `http://www.haskell.org/ghc/`.

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

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

R. Glück and M.H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Berlin: Springer-Verlag, 1996.

G. W. Hamilton. *Compile-Time Optimisation of Store Usage in Lazy Funtional Programs*. PhD thesis, University of Stirling, 1993.

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

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

J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

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

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

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

P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language, 2007. Submitted to IFL'07.

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

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

Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In Ralph Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages,*

and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, pages 473–486. ACM, 2005. ISBN 1-59593-031-0.

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

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

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.

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

M. Odersky and K. Läufer. Putting type annotations to work. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67. ACM, 1996.

A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 143–154. ACM, 2007. ISBN 1-59593-575-4.

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

A. Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1): 1–22, January 1998.

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

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 1996b.

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

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

J.P. Secher and M.B. 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.

O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

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

M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *International Logic Programming Symposium*, page to appear. Cambridge, MA: MIT Press, 1995.

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

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

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

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

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

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

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

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

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

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

P. Wadler. Listlessness is better than laziness. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52. ACM, ACM, August 1984.

P. Wadler. Listlessness is better than laziness II: composing listless functions. In H. Ganziger and N. D. Jones, editors, *Proc. Workshop on Programs as Data Objects*, volume 217 of *LNCS*. SpringerVerlag, 1985.

P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

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

K. Yoshii. Time-stamp counter, November 2007. URL `http://www-unix.mcs.anl.gov/~kazutomo/rdtsc.html`.