Time- and Size-Efficient Supercompilation

Peter A. Jonsson

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

Supervisor:

Johan Nordlander

To Moa

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. Supercompilation 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 supercompilation algorithms 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 positive supercompilation algorithms for higher-order call-by-value and call-by-name languages that preserves termination properties of the programs they optimize. We prove the call-by-value 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. We also suggest to speculatively supercompile expressions and discard the result if it turned out bad. To test this approach we implemented the call-by-name algorithm in GHC and performed measurements on the standard nofib benchmark suite. We manage to supercompile large parts of the imaginary and spectral parts of nofib in a matter of seconds while keeping the binary size increase below 5%.

Our algorithms are 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. This dissertation would not exist without his guidance and support over numerous years.

I spent the summer of 2009 at Microsoft Research in Cambridge working on supercompiling Haskell. Simon Peyton Jones spent a lot of time guiding me and has had a very strong influence on this work. Many of the ideas described in this thesis are initially his and he has contributed numerous ideas for improvements of the presentation as well.

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 first three years deserves a special mention – he has read many of my drafts and we have discussed numerous ideas over the years. Martin Kero and Andrey Kruglyak has also been great sources of ideas and inspiration.

I also spent three months in the functional programming laboratory at the university of Nottingham with Henrik Nilsson as host. It was a great experience and needless to say I know a lot more about dependent types, category theory, and functional reactive programming thanks to that time.

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.

The help I have gotten over the years from people that I have no formal association with must be acknowledged. I have spent many hours discussing program transformations with the following people, in no particular order: Colin Runciman, Neil Mitchell, Max Bolingbroke, Ilya Klyuchnikov, Neil Jones, Torben Mogensen, and Duncan Coutts. You have all taught me a great deal of technical things, as well as provided inspiration and ideas, and I am truly grateful for that. Simon Marlow and Ian Lynagh answered countless questions about the internals of GHC which helped realize our prototype implementation.

Finally, I would like to thank my wife Moa for her constant love, support, and understanding.

This dissertation is partly based on previously published material (Jonsson and Nordlander 2009, 2010a,b, 2011) and the work was funded by the AMI laboratory, the ESIS project and an internship at Microsoft Research.

CHAPTER 1

Introduction

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). Functional languages often cater to this need by providing a static type system and treating functions as first class citizens so they can be passed around freely higher-order functions. With a pure language (Sabry 1998) it is possible to automatically generate tests with tools such as QuickCheck (Claessen and Hughes 2000), and the programmer can perform equational reasoning about their programs if necessary.

Combining the above features is obviously attractive, but it is 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 2011. 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. The process of removing intermediate structures from programs is often called fusion or deforestation, and supercompilation is a particular family of algorithms that perform both fusion and program specialization. There has been plenty of research on fusion techniques in a call-by-name context, but it is only recently the problem has been considered in a call-by-value context (Ohori and Sasano 2007).

This is unfortunate since call-by-value languages are so common, and removing intermediate structures in a call-by-value language is perhaps even more important than in a lazy language since the entire intermediate structure has to remain in memory during the computation. Notable examples of call-by-value languages include OCaml (Leroy 2008), Standard ML (Milner et al. 1997) and F# (Syme 2008). Given that F# is currently being turned into a product it is unlikely that these languages will disappear any time soon.

To tackle the performance problems that come with intermediate structures and higher-order functions it has been suggested to apply existing fusion techniques for call-by-name languages to call-by-value languages, something that might convert looping programs into terminating programs. An example is the program

$$(\lambda x.y)\,(\mathit{fac}\,z)$$

which can loop if the value of z is negative. Applying Wadler's deforestation algorithm to the program will result in y, which is sound under call-by-name or call-by-need. Under call-by-value the non-termination in the original program has been removed, and hence the meaning of the program has been altered by the transformation.

This is no longer necessary, as this thesis develops a supercompilation algorithm that remove many of these intermediate structures and higher-order functions at compile time. With a small adjustment it can be applied to either a call-by-name, or a call-by-value language. We demonstrate that a naïve conversion of previous algorithms to call-by-value semantics yields worse results than necessary, and show how our call-by-value algorithm work well for examples from the literature that were originally intended to show the usefulness of call-by-name algorithms. We prove our call-by-value algorithm correct and implement it in an experimental compiler for Timber (Nordlander et al. 2008), a pure object-oriented functional language. We also implement our call-by-name algorithm in GHC to perform measurements on the standard nofib benchmark suite (Partain 1992). Our measurements show that certain programs can be sped up by an order of magnitude, and that several programs in nofib run 20-50% faster. We also provide a number of adjustments to the algorithm to mitigate the problems of both long compilation times and code explosion resulting in large binaries. Our work is a necessary first step towards supercompiling impure call-by-value languages.

1.1 General Outline

Chapters 2 to 5 are the central core of this dissertation.

Chapter 1: Introduction We introduce the problem and motivate why it is an interesting problem.

Chapter 2: Higher Order Supercompilation Two algorithms that specializes programs and automatically removes intermediate data structures for higher-order languages are presented.

Chapter 3: Algorithm Correctness We prove our supercompilation algorithm for higher-order call-by-value languages is correct.

Chapter 4: Controlling and Extending Supercompilation We introduce the idea of speculatively transforming expressions and discard bad results in order to mitigate the problem of code explosion. We also present additional pre- and post-conditions for the rules in our algorithm to improve the performance.

Chapter 5: Implementation and Measurements Measurements from a set of common examples from the literature on program transformations. We show that our positive supercompiler does remove intermediate structures, and can improve the performance by an order of magnitude for certain benchmarks. We also show that we can supercompile large parts of the imaginary and spectral parts of the nofib benchmark suite in a matter of seconds while keeping the binary size increase below 5%.

Chapter 6: Related Work A survey of the state of the art within the field of program transformations that perform specialization and intermediate data structure removal.

Chapter 7: Conclusions and Future Research Conclusions are drawn from the work presented in previous chapters, a summary of contributions of this work and a discussion of further work.

Appendix A: Function Definitions We define standard prelude functions that are used in the examples.

Appendix B: Proofs Long proofs for theorems and lemmas used in Chapter 3.

1.2 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:

```
pow x y = if y = 0 then
              1
          else
              x * pow x (y - 1)
```

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:

```
filter p xs = [ x | x <- xs, p x ]
```

$$lowers\ xs\ = filter\ isLower\ xs$$

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

$$filter_{isLower}\ xs = [\, x \mid x\ <-\ xs,\ isLower\ x\,]$$

$$lowers\ xs \qquad = filter_{isLower}\ xs$$

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.

1.3 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:

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

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])

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 call-by-need language – complete intermediate structures need to stay on the heap since they

```
natural          = reverse . map ('mod' 10) . takeWhile (≠ 0) . iterate ('div' 10)

fastNatural n = fastNatural' n []
  where
    fastNatural' n m = if n ≠ 0 then
                           fastNatural' (n 'div' 10) (n 'mod' 10 : m)
                         else
                           m
```

Figure 1.2: Obtain a list of a number's 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

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.

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 $|xs| + |ys|$. We start with a step by step example transformation in a call-by-name language to give the reader a better intuition on the small transformation steps the algorithm uses:

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

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

```
letrec $h_1\ xs'\ ys'\ zs'$ = case ( case $xs'$ of
                                          $[] \to ys'$
                                          $(x_1 : xs_1) \to x_1 : append\ xs_1\ ys'$) of
                                     $[] \to zs'$
                                     $(x : xs) \to x : append\ xs\ zs'$
  in $h_1\ xs'\ ys'\ zs'$
```

Pushing down the outer case-expression into both branches of the inner one leads to

```
letrec $h_1\ xs'\ ys'\ zs'$ = case $xs'$ of
                                     $[] \to$ case $ys'$ of
                                          $[] \to zs'$
                                          $(x : xs) \to x : append\ xs\ zs'$
                                     $(x_1 : xs_1) \to$ case $x_1 : append\ xs_1\ ys'$ of
                                               $[] \to zs'$
                                               $(x : xs) \to x : append\ xs\ zs'$
  in $h_1\ xs'\ ys'\ zs'$
```

Alpha-converting the first branch and reducing the resulting case-expression of a known constructor

```
letrec $h_1\ xs'\ ys'\ zs'$ = case $xs'$ of
                                     $[] \to$ case $ys'$ of
                                          $[] \to zs'$
                                          $(y : ys) \to y : append\ ys\ zs'$
                                     $(x_1 : xs_1) \to x_1 : append\ (append\ xs_1\ ys')\ zs'$
  in $h_1\ xs'\ ys'\ zs'$
```

Transform each branch separately. Transformation of the second branch in the first branch will create a new function h_2 that is isomorphic to *append*, and the second branch of the outer case is a renaming of our initial expression called h_1, so we insert a call to that. The final result is:

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

We can see that h_1 will only traverse *xs'* once and then call h_2 to append the tail of *ys'* and *zs'*, allowing us to conclude that that the new runtime is indeed $|xs'| + |ys'|$. The next section shows, with a similar step by step transformation on this example, how we safely can obtain the same result for a call-by-value language by making a small adjustment to one of the transformation steps. Chapter 2 contains the complete algorithm with all the necessary technical details.

1.4 Preserving Semantics

We already saw an example in the introduction how applying Wadler's deforestation algorithm to some programs could alter the meaning of these programs in a call-by-value context. 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 algorithms.

One might think that it is sufficient to modify a call-by-name algorithm to simply delay beta reduction until every function argument has been specialized to a value. However, it turns out that this strategy misses even simple opportunities to remove intermediate structures. That is, eager specialization of function arguments risks destroying *fold* opportunities that might otherwise appear, something which may prohibit complexity improvements to the resulting program.

The novelty of our supercompilation algorithm is that it concentrates all call-by-value dependencies to a single rule that relies on the result from a separate strictness analysis for correct behavior. In effect, our transformation delays transformation of function arguments past inlining, much like a call-by-name scheme does, although only as far as allowed by call-by-value semantics. The result is an algorithm that is able to improve a wide range of illustrative examples like the existing algorithms do, but without the risk of introducing artificial termination.

We saw an example in Section 1.3 of how Wadler's deforestation algorithm transforms $append\ (append\ xs\ ys)\ zs$ so that it saves one traversal of the first list, thereby reducing the complexity from $2|xs| + |ys|$ to $|xs| + |ys|$.

If we naïvely change Wadler's algorithm to call-by-value semantics by eagerly attempting to transform arguments before attacking the body, we do not achieve this improvement. We give an example of a hypothetical call-by-value variant of Wadler's deforestation algorithm that attacks arguments first:

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

Naming this expression h_1 and inlining the body of the inner *append* and then pushing down the outer call into each branch gives

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

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

$$\begin{array}{l}
\textbf{letrec}\ h_1\ xs'\ ys'\ zs' \ = \ \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [] \ \rightarrow \ h_2\ ys'\ zs' \\
\qquad\qquad\qquad\qquad\qquad (x : xs) \ \rightarrow \ [x : append\ xs\ ys'/z]append\ z\ zs' \\
\qquad\quad h_2\ ys\ zs' \ = \ \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \ \rightarrow \ zs' \\
\qquad\qquad\qquad\qquad (y' : ys') \ \rightarrow \ y' : h_2\ ys'\ zs' \\
\textbf{in}\ h_1\ xs'\ ys'\ zs'
\end{array}$$

Both these expressions are renamings of $append\ ys'\ zs'$ which already has a name, h_2, so insert a call to that function:

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

Performing the substitution gives us the final result:

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

Notice that the intermediate structure from the input program is still there after the transformation, and the complexity is still $2|xs| + |ys|$! This can be compared to how the same example is transformed by Wadler's algorithm as shown in Section 1.3. The reason our hypothetical call-by-value algorithm failed to improve the program is that it had to split expressions too early during the transformation, thereby preventing fold opportunities that occur in a call-by-name setting.

However, changing the call-by-value algorithm to do the exact opposite — that is, carefully delaying the transformation of arguments to a function past the inlining of its body, but only as far as strictness allows — actually leads to the same result that Wadler obtains with $append\ (append\ xs\ ys)\ zs$. This is a key observation for obtaining deforestation under call-by-value without altering the semantics, and our transformation exploits it.

Except for the fundamental reliance on strictness analysis, which is necessary to preserve semantics, our transformation shares many of its rules with Wadler's algorithm.

CHAPTER 2

Higher Order Supercompilation

This chapter contains two supercompilation algorithms designed for program optimization: one for a call-by-name language and one for a call-by-value language. We revise the standard design of supercompilation algorithms and sacrifice some in order to have a faster algorithm. The performance rests on two important design choices:

Term Representation: We represent terms as zippers (Huet 1997) during the transformation, which gives efficient and convenient implementations of tests for non-termination and splitting of terms. As a side effect this representation also makes it easy to reduce the number of terms to test for non-termination: only test against those terms who have the same head as the current term.

Term Size in Focus: By decreasing the size of the term that our supercompiler focuses on we can effectively decrease the cost of testing for non-termination. This shows in our compilation times: they are less than 3 seconds on a modern desktop computer for all the small examples from the nofib suite (Partain 1992).

2.1 Language

Our language of study is a higher-order functional language with recursive function names, let-expressions, and case-expressions. All expressions must be typable under System F, and we explicitly exclude negative data types. Its syntax for expressions, values and patterns is shown in Figure 2.1.

Here we let variables and constructor symbols be denoted by *x* and *k*, respectively. The constructor symbols *k* range over a set *K* and we also assume that there is a separate set $\mathcal{G}$ of recursively defined function symbols, ranged over by g. In what follows we will assume that the meaning of such symbols is accessed by $g \overset{\text{def}}{=} e$.

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 $\oplus$.

Expressions

$$\begin{array}{lll} e & ::= & n \mid x \mid g \mid e\,e' \mid \lambda x.e \mid k\,\overline{e} \mid e_1 \oplus e_2 \mid \textbf{case}\,e\ \textbf{of}\ \{p_i \rightarrow e_i\} \\ & \mid & \textbf{let}\,x = e\ \textbf{in}\,e' \end{array}$$

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

Values

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

Weak Head Normal Form

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

Figure 2.1: The language

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

Figure 2.2: Free variables of an expression

We denote the free variables of an expression *e* by $\mathit{fv}(e)$, as defined in Figure 2.2 and lists of expressions $e_1 \ldots e_n$ and variables are $x_1 \ldots x_n$ denoted as $\overline{e}$ and $\overline{x}$.

A program is an expression with no free variables and all function names defined in *G*. The intended operational call-by-value semantics is given in Figure 2.3 and the call-by-name semantics in Figure 2.4, where $[\overline{e}/\overline{x}]e'$ is the capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e'.

A reduction context $\mathcal{E}$ is a term containing a single hole, $\square$, which indicates the next expression to be reduced. The expression $[e]\mathcal{E}$ is the term obtained by replacing the hole in $\mathcal{E}$ with e. $\overline{\mathcal{E}}$ denotes a list of terms with just a single hole, evaluated from left to right. $\equiv$ is equivalence between expressions up to alpha-conversion.

If a variable appears no more than once in an expression, that expression is said to be *linear* with respect to that variable. Like Wadler (1990), we extend the definition slightly for linear case-expressions: no variable may appear in both the head and a branch, although a variable may appear in more than one branch. For example, the definition of *append* is linear with respect to *ys*, although *ys* appears in both branches.

Call-by-value reduction contexts

$$\begin{array}{lll} \mathcal{E} & ::= & \square \mid \mathcal{E}\,e \mid (\lambda x.e)\,\mathcal{E} \mid k\,\overline{\mathcal{E}} \mid \mathcal{E} \oplus e \mid n \oplus \mathcal{E} \mid \textbf{case}\ \mathcal{E}\ \textbf{of}\ \{p_i \rightarrow e_i\} \\ & \mid & \textbf{let}\, x = \mathcal{E}\ \textbf{in}\, e \end{array}$$

Call-by-value evaluation relation

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

Figure 2.3: Call-by-value reduction semantics

Call-by-name reduction contexts

$$\mathcal{N} \quad ::= \quad \square \mid \mathcal{N}\,e \mid \mathcal{N} \oplus e \mid n \oplus \mathcal{N} \mid \textbf{case}\ \mathcal{N}\ \textbf{of}\ \{p_i \rightarrow e_i\}$$

Call-by-name evaluation relation

$$\begin{array}{llll} [g]\mathcal{N} & \mapsto_n & [e]\mathcal{N}, \text{ if } g \overset{\text{def}}{=} e & \text{(Global)} \\ [(\lambda x.e)\, e']\mathcal{N} & \mapsto_n & [[e'/x]e]\mathcal{N} & \text{(Beta)} \\ [\textbf{let}\, x = e\, \textbf{in}\, e']\mathcal{N} & \mapsto_n & [[e/x]e']\mathcal{N} & \text{(Let)} \\ [\textbf{case}\, k\, \overline{e}\, \textbf{of}\ \{k_i\, \overline{x}_i \rightarrow e_i\}]\mathcal{N} & \mapsto_n & [[\overline{e}/\overline{x}_j]e_j]\mathcal{N}, \text{ if } k = k_j & \text{(KCase)} \\ [\textbf{case}\, n\, \textbf{of}\ \{n_i \rightarrow e_i\}]\mathcal{N} & \mapsto_n & [e_j]\mathcal{N}, \text{ if } n = n_j & \text{(NCase)} \\ [n_1 \oplus n_2]\mathcal{N} & \mapsto_n & [n]\mathcal{N}, \text{ if } n = n_1 + n_2 & \text{(Arith)} \end{array}$$

Figure 2.4: Call-by-name reduction semantics

2.2 Higher Order Positive Supercompilation

Our supercompiler is defined as two mutually recursive functions that pattern-match on expressions and contexts in order to rewrite expressions. The first algorithm is called the *driving* algorithm, used to perform evaluation steps, and the second algorithm is called the *building* algorithm, which reassembles the transformed expression. These algorithms are defined in Figure 2.6. Both algorithms take three parameters: 1) an expression; 2) a memoization list ρ modelled by a substitution; and 3) the context $\mathcal{R}$. The memoization list holds information about expressions already traversed, and their generated fresh names, and is used to guarantee termination. A notational convention is to use e' to denote expressions that are already transformed.

We replace our nested evaluation contexts with a list of shallow reduction contexts for our supercompiler as defined in Figure 2.5. It will turn out to be convenient for the implementation

The outer context:

$$\mathcal{R} ::= \epsilon \mid \mathcal{F} : \mathcal{R}$$

Each frame:

$$\mathcal{F} ::= \square\, e \mid \square \oplus e \mid n \oplus \square \mid \textbf{case} \ \square \ \textbf{of} \ \{p_i \to e_i\}$$

Memoization element:

$$\nu ::= (h, \lambda \overline{x}.[g]\mathcal{R})$$

Memoization list:

$$\rho ::= \epsilon \mid \nu : \rho$$

Figure 2.5: Context definition

to pattern match on the context if it is a list of single level contexts, known as a zipper (Huet 1997). We let $\overline{\square\, e}$ denote a list of application contexts.

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R10-R13 are the default focusing rules that shift or extend the given driving context $\mathcal{R}$ and focus on the next expression to be transformed. If no other rule matches, such as for a constructor in the empty context, rule R14 will match and call build to rebuild the expression.

The program is turned "inside-out" by moving the surrounding context $\mathcal{R}$ into all branches of the case-expression through rules R18 and R19. Rule R9 has a similar mechanism for let-expressions. Rule R8 is only allowed to match if the variable *y* is not freshly generated by the splitting mechanism described in Section 2.2.2.

We use the operator $\rho|_g$ to extract all expressions with the head g from ρ. Our implementation stores the partitioned ρ's to avoid repeating the partitioning. Partitioning the memoization list gives significant time savings, but it can only be applied on the outermost level.

Rule R3 refers to $\mathcal{D}_{app}(\,)$ which is defined in Figure 2.7. $\mathcal{D}_{app}(\,)$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation. Whenever an expression that is equivalent up to renaming of variables to a previous application, a call to the associated function symbol is inserted instead. This is not sufficient to guarantee termination of the algorithm, but the mechanism is crucial for the complexity improvements mentioned in Section 1.3.

2.2.1 Termination Checking

To ensure termination of our supercompiler we need to define a predicate that is commonly referred to as "the whistle", $\trianglelefteq$, in literature on supercompilation. The intuition is that when $e \trianglelefteq e'$, *e'* contains all subexpressions of *e*, possibly embedded in other expressions. For any infinite sequence $e_0, e_1, \ldots$ there must exist an *i* and a *j* such that $i < j$ and $e_i \trianglelefteq e_j$. This condition is sufficient to ensure termination.

Whenever the whistle blows, our algorithm splits the input expression into strictly smaller expressions that are transformed separately in the empty context. This might expose new folding opportunities, and allows the algorithm to remove intermediate structures in subexpressions.

General form:
$\mathcal{D}\llbracket e \rrbracket \,\rho\, \mathcal{R} = e'$

Evaluation Rules

$$\begin{array}{lcll}
\mathcal{D}\llbracket n_j \rrbracket \,\rho\, (\textbf{case}\ \Box\ \ \textbf{of}\,\{n_i \rightarrow e_i\} : \mathcal{R}) & = & \mathcal{D}\llbracket e_j \rrbracket \,\rho\, \mathcal{R} & \text{(R1)} \\
\mathcal{D}\llbracket n \rrbracket \,\rho\, (n_1 \oplus \Box : \mathcal{R}) & = & \mathcal{D}\llbracket n_2 \rrbracket \,\rho\, \mathcal{R}\text{, where } n_2 = n_1 + n & \text{(R2)} \\
 & & & \\
\mathcal{D}\llbracket g \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{D}_{app}(g) \,\rho\, \mathcal{R} & \text{(R3)} \\
 & & & \\
\mathcal{D}\llbracket k_j \rrbracket \,\rho\, (\overline{\Box\, e} : \textbf{case}\ \Box\ \ \textbf{of}\,\{k_i\, \overline{x}_i \rightarrow e_i\} : \mathcal{R}) & = & \mathcal{D}\llbracket \textbf{let}\, \overline{x}_j = \overline{e}\, \textbf{in}\, e_j \rrbracket \,\rho\, \mathcal{R} & \text{(R4)} \\
 & & & \\
\mathcal{D}\llbracket \lambda \overline{x}.e_1 \rrbracket \,\rho\, (\overline{\Box\ e_2} : \mathcal{R}) & = & \mathcal{D}\llbracket \textbf{let}\, \overline{x} = \overline{e_2}\, \textbf{in}\, e_1 \rrbracket \,\rho\, \mathcal{R} & \text{(R5)} \\
\mathcal{D}\llbracket \lambda x.e \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{B}\llbracket \lambda x.(\mathcal{D}\llbracket e \rrbracket \,\rho\, \epsilon) \rrbracket \,\rho\, \mathcal{R} & \text{(R6)} \\
 & & & \\
\mathcal{D}\llbracket \textbf{let}\, x = n\, \textbf{in}\, e \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{D}\llbracket [n/x]e \rrbracket \,\rho\, \mathcal{R} & \text{(R7)} \\
\mathcal{D}\llbracket \textbf{let}\, x = y\, \textbf{in}\, e \rrbracket \,\rho\, \mathcal{R} \mid y \text{ not freshly generated} & = & \mathcal{D}\llbracket [y/x]e \rrbracket \,\rho\, \mathcal{R} & \text{(R8)} \\
\mathcal{D}\llbracket \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \rrbracket \,\rho\, \mathcal{R} \mid x \in \textit{linear}(e_2) & = & \mathcal{D}\llbracket [e_1/x]e_2 \rrbracket \,\rho\, \mathcal{R} & \text{(R9)} \\
\qquad\qquad\qquad\qquad\quad \mid \text{otherwise} & = & \textbf{let}\, x = \mathcal{D}\llbracket e_1 \rrbracket \,\rho\, \epsilon\, \textbf{in}\, \mathcal{D}\llbracket e_2 \rrbracket \,\rho\, \mathcal{R} &
\end{array}$$

Focusing Rules

$$\begin{array}{lcll}
\mathcal{D}\llbracket n \rrbracket \,\rho\, (\Box\ \oplus e_2 : \mathcal{R}) & = & \mathcal{D}\llbracket e_2 \rrbracket \,\rho\, (n \oplus \Box : \mathcal{R}) & \text{(R10)} \\
\mathcal{D}\llbracket e_1 \oplus e_2 \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{D}\llbracket e_1 \rrbracket \,\rho\, (\Box\ \oplus e_2 : \mathcal{R}) & \text{(R11)} \\
\mathcal{D}\llbracket e_1\, e_2 \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{D}\llbracket e_1 \rrbracket \,\rho\, (\Box\ e_2 : \mathcal{R}) & \text{(R12)} \\
\mathcal{D}\llbracket \textbf{case}\, e\ \textbf{of}\,\{p_i \rightarrow e_i\} \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{D}\llbracket e \rrbracket \,\rho\, (\textbf{case}\ \Box\ \ \textbf{of}\,\{p_i \rightarrow e_i\} : \mathcal{R}) & \text{(R13)}
\end{array}$$

Fallthrough

$$\begin{array}{lcll}
\mathcal{D}\llbracket e \rrbracket \,\rho\, \mathcal{R} & = & \mathcal{B}\llbracket e \rrbracket \,\rho\, \mathcal{R} & \text{(R14)}
\end{array}$$

Rebuilding Expressions

$$\begin{array}{lcll}
\mathcal{B}\llbracket e' \rrbracket \,\rho\, (\Box\ \oplus e_2 : \mathcal{R}) & = & \mathcal{B}\llbracket e' \oplus (\mathcal{D}\llbracket e_2 \rrbracket \,\rho\, \epsilon) \rrbracket \,\rho\, \mathcal{R} & \text{(R15)} \\
\mathcal{B}\llbracket e' \rrbracket \,\rho\, (n \oplus \Box : \mathcal{R}) & = & \mathcal{B}\llbracket n \oplus e' \rrbracket \,\rho\, \mathcal{R} & \text{(R16)} \\
\mathcal{B}\llbracket e' \rrbracket \,\rho\, (\Box\, e : \mathcal{R}) & = & \mathcal{B}\llbracket e' (\mathcal{D}\llbracket e \rrbracket \,\rho\, \epsilon) \rrbracket \,\rho\, \mathcal{R} & \text{(R17)} \\
\mathcal{B}\llbracket x' \rrbracket \,\rho\, (\textbf{case}\ \Box\ \ \textbf{of}\,\{p_i \rightarrow e_i\} : \mathcal{R}) & = & \textbf{case}\, x'\ \textbf{of}\,\{p_i \rightarrow \mathcal{D}\llbracket [p_i/x']e_i \rrbracket \,\rho\, ([p_i/x']\mathcal{R})\} & \text{(R18)} \\
\mathcal{B}\llbracket e' \rrbracket \,\rho\, (\textbf{case}\ \Box\ \ \textbf{of}\,\{p_i \rightarrow e_i\} : \mathcal{R}) & = & \textbf{case}\, e'\, \textbf{of}\,\{p_i \rightarrow \mathcal{D}\llbracket e_i \rrbracket \,\rho\, \mathcal{R}\} & \text{(R19)} \\
\mathcal{B}\llbracket e' \rrbracket \,\rho\, \epsilon & = & e' & \text{(R20)}
\end{array}$$

Figure 2.6: Driving algorithm and building algorithm

$$
\begin{array}{lll}
\mathcal{D}_{app}(g)\,\rho\,\mathcal{R} \mid \exists h.\rho|_g(h) \equiv \lambda\overline{x}.[g]\mathcal{R} & = h\,\overline{x} & (1) \\
\qquad\qquad \mid \exists h.\rho|_g(h) \trianglelefteq \lambda\overline{x}.[g]\mathcal{R} & & (2) \\
\qquad\qquad ,(f_g, [\overline{f}/\overline{y}]) = \mathit{divide}(\lambda\overline{x}.[g]\mathcal{R}, \rho(h)) & = [\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon/\overline{y}]\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon & \\
\qquad\qquad \mid \text{otherwise} & = \textbf{letrec}\,h = \lambda\overline{x}.\mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R}\,\textbf{in}\,h\,\overline{x} & (3) \\
\quad \text{where } g \overset{\text{def}}{=} e & & \\
\qquad\quad \rho' = (h, \lambda\overline{x}.[g]\mathcal{R}) : \rho & & \\
\qquad\quad h \text{ fresh} & & \\
\qquad\quad \overline{x} = \mathit{fv}([g]\mathcal{R}) & &
\end{array}
$$

Figure 2.7: Driving of applications

We need a definition of uniform terms analogous to the one defined by Sørensen and Glück (1995) to define our whistle. We slightly adjust their version to fit our language.

Definition 2.1 (Uniform terms)**.** *Let s range over the set* $\mathcal{G} \cup K \cup \{\mathbf{caseof}, \mathbf{let}, \mathbf{letrec}, \mathbf{primop}, \mathbf{lambda}, \mathbf{apply}\}$*, and let* $\mathbf{caseof}(\overline{e}), \mathbf{let}(\overline{e}), \mathbf{letrec}(\overline{v}, e), \mathbf{primop}(\overline{e}), \mathbf{lambda}(e)$*, and* $\mathbf{apply}(\overline{e})$ *denote a case, let, recursive let, primitive operation, lambda abstraction or application for all subexpressions* $\overline{e}, e$ *and* $\overline{v}$*. The set of terms T is the smallest set of arity respecting symbol applications* $s(\overline{e})$*.*

Definition 2.2 (Homeomorphic embedding)**.** *Define* $\trianglelefteq$ *as the smallest relation on* T *satisfying:*

$$
x \trianglelefteq y, \qquad n_1 \trianglelefteq n_2, \qquad \frac{e \trianglelefteq e_i \textit{ for some } i}{e \trianglelefteq s(e_1, \ldots, e_n)}, \qquad \frac{e_1 \trianglelefteq e'_1, \ldots, e_n \trianglelefteq e'_n}{s(e_1, \ldots, e_n) \trianglelefteq s(e'_1, \ldots, e'_n)}
$$

2.2.2 Expression Splitting

Whenever the whistle blows, our transformation splits the input expression into strictly smaller terms that are transformed separately in the empty context. It is safe to just leave the current expression in the output program, but by splitting the expression and transforming the parts separately it is possible to remove intermediate structures in subexpressions. The design follows the positive supercompilation algorithm outlined by Sørensen (2000), except that we reassemble the transformed subexpressions into an expression of the original form instead of pulling them out as let-definitions. This will turn out to be convenient in order to preserve strictness for the call-by-value algorithm presented in Section 2.3. Our transformation is also more complicated because we perform the program extraction immediately, rather than constructing a process tree and then extracting the program in a separate pass as done by Turchin (1986b) and many others.

Divide tries to preserve as much structure between its first input argument and output by first calling an adaption of the most specific generalisation (*msg*) (Sørensen and Glück 1995) to the zipper representation. Our adaption is a bit unconventional in that it returns an expression and a substitution rather than the standard of an expression and two substitutions. Should the call to msg fail, divide instead calls split which will always return a valid result under the conditions ensured by the driving algorithm: divide is always called on two expressions on the form $\lambda\overline{x}.[g]\mathcal{R}$ and $\lambda\overline{x}.[g]\mathcal{R}'$. Divide, split and msg are defined in Figure 2.8. For the

$$
\begin{array}{ll}
divide(\lambda\overline{x}.[e_1]\mathcal{R}, \lambda\overline{x}.[e_2]\mathcal{R}') = (e,\theta) & \text{iff } (e,\theta) = msg([e_1]\mathcal{R}, [e_2]\mathcal{R}') \\
 & \text{and } sizeof(e) < sizeof([e_1]\mathcal{R}) \\[1em]
divide(\lambda\overline{x}.[e_1]\mathcal{R}, \lambda\overline{x}.[e_2]\mathcal{R}') = split([e_1]\mathcal{R}) & \\[1em]
split([g]\mathcal{F}_1 : \ldots : \mathcal{F}_n : \epsilon) = ([x]\mathcal{F}_n : \epsilon, [([g]\mathcal{F}_1 : \ldots : \mathcal{F}_{n-1} : \epsilon)/x]) \; x \text{ fresh} & \\[1em]
msg([e_1]\mathcal{R}_1, [e_2]\mathcal{R}_2) = ([e]\mathcal{R}, \theta\theta') & \text{if } \mathcal{R}' = \epsilon \\
\qquad\qquad = ([x]\mathcal{R}', \theta\theta'[[e]\mathcal{R}/x]) & \text{otherwise} \\
\qquad \text{if } (e,\theta) = msg(e_1,e_2) \text{ and } (\mathcal{R},\mathcal{R}',\theta') = msg(\mathcal{R}_1,\mathcal{R}_2) & \\[1em]
msg(\Box\, e_1, \Box\, e_2) = (\Box\, e', \theta) & \text{iff } (e',\theta) = msg(e_1,e_2) \\
msg(\Box \oplus e_1, \Box \oplus e_2) = (\Box \oplus e', \theta) & \text{iff } (e',\theta) = msg(e_1,e_2) \\
msg(e_1 \oplus \Box, e_2 \oplus \Box) = (e' \oplus \Box, \theta) & \text{iff } (e',\theta) = msg(e_1,e_2) \\
msg(\mathbf{case}\ \Box\ \mathbf{of}\{p_i \rightarrow e_{1i}\}, & \\
\qquad \mathbf{case}\ \Box\ \mathbf{of}\{p_i \rightarrow e_{2i}\}) = (\mathbf{case}\ \Box\ \mathbf{of}\{p_i \rightarrow e_i'\}, \theta_1 \ldots \theta_i) & \text{iff } (e_i',\theta_i) = msg(e_{1i},e_{2i}) \\[1em]
msg(\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2, \mathbf{let}\, x = e_3\, \mathbf{in}\, e_4) = (\mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2', \theta\theta') & \text{iff } (e_1',\theta) = msg(e_1,e_3) \\
 & \text{and } (e_2',\theta') = msg(e_2,e_4) \\
msg(\lambda x.e_1, \lambda x.e_2) = (\lambda x.e', \theta) & \text{iff } (e',\theta) = msg(e_1,e_2) \\
msg(g,g) = (g,\emptyset) & \\
msg(k,k) = (k,\emptyset) & \\
msg(n,n) = (n,\emptyset) & \\
msg(x,x) = (x,\emptyset) & \\
msg(e_1,e_2) = (x,[e_1/x]) & x \text{ fresh} \\[1em]
msg(\mathcal{F}_1 : \mathcal{R}_1, \mathcal{F}_2 : \mathcal{R}_2) = (\mathcal{F} : \mathcal{R}, \mathcal{R}', \theta\theta') & \\
\qquad \text{if } (\mathcal{F},\theta) = msg(\mathcal{F}_1,\mathcal{F}_2) \text{ and } (\mathcal{R},\mathcal{R}',\theta') = msg(\mathcal{R}_1,\mathcal{R}_2) & \\[1em]
msg(\mathcal{R}_1,\mathcal{R}_2) = (\epsilon, \mathcal{R}_1, \emptyset) & \text{(default)}
\end{array}
$$

Figure 2.8: Divide, split and msg in zipper form

e		e’	t_g	θ
e	$\trianglelefteq$	$Just\ e$	x	$[e/x]$
$Right\ e$	$\trianglelefteq$	$Right\ (e,\ e')$	$Right\ x$	$[e/x]$
$fac\ y$	$\trianglelefteq$	$fac\ (y\ -\ 1)$	$fac\ x$	$[y/x]$

Figure 2.9: Examples of the whistle and msg

presentation we ignore the issues concerned with splitting under binders in expressions since binders are treated in the same way as described by Pfenning (1991).

The *msg* can return an expression which is of greater size than its input because of binders: when splitting the two expressions **case** x **of** $\{\ (a,\ b)\ \rightarrow\ a\ \}$ and **case** x **of** $\{\ (a,\ b)\ \rightarrow\ b\ \}$ the result will be (**case** x **of** $\{\ (a,\ b) \rightarrow\ z\ a\ \},\ [(\lambda a.a)/z])$.

All the examples of how our transformation works in Chapter 1 and Section 2.3 eventually terminate through a combination of alternative 1 and alternative 3 of $\mathcal{D}_{app}()$. Alternative 2 is used when expressions are "growing" in some sense, and a specific example is *reverse* with an accumulating parameter is shown in Figure 2.10.

2.3 Supercompiling Call-by-Value Languages

We claimed in Chapter 1 that our transformation compares favorably with previous call-by-name transformations, and we now proceed with demonstrating the transformation on some common examples. The results of the transformation on these examples are identical to the results of Wadler's deforestation algorithm (Wadler 1990). Unfortunately this does not hold in general as a counter-example is the transformation of the expression $zip\ (map\ f\ xs)\ (map\ g\ ys)$ where Wadler's algorithm will eliminate both intermediate structures and our transformation will only eliminate the first intermediate structure. We get back to this example in Section 4.8.1 and show how to strengthen our algorithm to eliminate both intermediate structures.

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

$$\begin{array}{lcl}
\textbf{letrec}\ h_0\ ys & = & \textbf{case}\ map\ square\ ys\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow x' + sum\ xs' \\
\textbf{in}\ h_0\ ys & &
\end{array}$$

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

$$\begin{array}{lcl}
\textbf{letrec}\ h_0\ ys & = & \textbf{case}\ (\ \textbf{case}\ ys\ \textbf{of} \\
 & & \qquad\qquad [] \rightarrow [] \\
 & & \qquad\qquad (x' : xs') \rightarrow (square\ x') : map\ square\ xs')\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow x' + sum\ xs' \\
\textbf{in}\ h_0\ ys & &
\end{array}$$

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

$$\begin{array}{lcl}
\textbf{letrec}\ h_0\ ys & = & \textbf{case}\ ys\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow square\ x' + sum\ (map\ square\ xs') \\
\textbf{in}\ h_0\ ys & &
\end{array}$$

At this point we inline the body of the first *square* occurrence:

$$\begin{array}{lcl}
\textbf{letrec}\ h_0\ ys & = & \textbf{case}\ ys\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow h_1\ x'\ xs'
\end{array}$$

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

(By rule 3 of $\mathcal{D}_{app}()$, put $(h_0, \lambda xs.rev\ xs\ [])$ in ρ and transform the program according to the rules of the algorithm)

$$\begin{array}{l}
\textbf{letrec}\ h_0\ xs\ =\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \rightarrow [] \\
\qquad\qquad\qquad\quad (x' : xs') \rightarrow \mathcal{D}[\![rev\ xs'\ (x' : [])]\!] \\
\ \textbf{in}\ h_0\ xs
\end{array}$$

(Recall that ρ contains $\lambda xs.rev\ xs\ []$ so alternative 2 of $\mathcal{D}_{app}(\,)$ is triggered and the expression is generalized against $rev\ xs\ []$)

=

$$\begin{array}{l}
\textbf{letrec}\ h_0\ xs\ =\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \rightarrow [] \\
\qquad\qquad\qquad\quad (x' : xs') \rightarrow [\mathcal{D}[\![(x' : [])]\!]/zs]\mathcal{D}[\![rev\ xs'\ zs]\!] \\
\ \textbf{in}\ h_0\ xs
\end{array}$$

=

$$\begin{array}{l}
\textbf{letrec}\ h_0\ xs\ =\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \rightarrow [] \\
\qquad\qquad\qquad\quad (x' : xs') \rightarrow [(x' : [])/zs]\mathcal{D}[\![rev\ xs'\ zs]\!] \\
\ \textbf{in}\ h_0\ xs
\end{array}$$

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

=

$$\begin{array}{l}
\textbf{letrec}\ h_0\ xs\ =\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \rightarrow [] \\
\qquad\qquad\qquad\quad (x' : xs') \rightarrow h_1\ xs'\ (x' : []) \\
\qquad\qquad h_1\ xs\ zs\ =\ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \rightarrow zs \\
\qquad\qquad\qquad\qquad (x' : xs') \rightarrow h_1\ xs'\ (x' : zs) \\
\ \textbf{in}\ h_0\ xs
\end{array}$$

Figure 2.10: Example of downwards generalization

$$\begin{array}{l} \qquad h_1\ x'\ xs' \ = \ x' * x' + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ xs') \\ \textbf{in}\ h_0\ ys \end{array}$$

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

$$\begin{array}{l} \textbf{letrec}\ h_0\ ys \ = \ \textbf{case}\ ys\ \textbf{of} \\ \qquad\qquad\qquad\quad [] \rightarrow 0 \\ \qquad\qquad\qquad\quad (x' : xs') \rightarrow h_1'\ x'\ xs' \\ \qquad h_1\ x'\ xs' \ = \ x' * x' + h_0\ xs' \\ \textbf{in}\ h_0\ ys \end{array}$$

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

The work by Ohori and Sasano (2007) cannot fuse two successive applications of the same function, nor mutually recursive functions. We show in the next two examples that our transformation can handle these cases. We need the following new function definitions:

$$\begin{array}{lll} \mathit{mapsq}\ xs & = & \textbf{case}\ xs\ \textbf{of} \\ & & \quad [] \rightarrow [] \\ & & \quad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs' \\ f\ xs & = & \textbf{case}\ xs\ \textbf{of} \\ & & \quad [] \rightarrow [] \\ & & \quad (x' : xs') \rightarrow (2 * x') : g\ xs' \\ g\ xs & = & \textbf{case}\ xs\ \textbf{of} \\ & & \quad [] \rightarrow [] \\ & & \quad (x' : xs') \rightarrow (3 * x') : f\ xs' \end{array}$$

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

$$\begin{array}{l} \textbf{letrec}\ h_2\ xs \ = \ h_3\ xs \\ \qquad h_3\ xs \ = \ \textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad [] \rightarrow [] \\ \qquad\qquad\qquad\qquad\qquad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs')\ \textbf{of} \\ \qquad\qquad\qquad\quad [] \rightarrow [] \\ \qquad\qquad\qquad\quad (x' : xs') \rightarrow (x' * x') : \mathit{mapsq}\ xs' \\ \textbf{in}\ h_2\ xs \end{array}$$

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

$$\textbf{letrec}\ h_2\ xs \ = \ h_3\ xs$$

$$\begin{array}{lll}
\quad h_3\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [] \rightarrow [] \\
 & & \quad (x' : xs') \rightarrow (x' * x' * x' * x') : mapsq\ (mapsq\ xs') \\
\textbf{in}\ h_2\ xs & &
\end{array}$$

Here we encounter a similar expression to what we started with, and create a new function call to h_2. The final result of our transformation are two new residual functions h_2 and h_3 that only traverses the input once:

$$\begin{array}{lll}
\textbf{letrec}\ h_2\ xs & = & h_3\ xs \\
\quad h_3\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [] \rightarrow [] \\
 & & \quad (x' : xs') \rightarrow (x' * x' * x' * x') : h_2\ xs' \\
\textbf{in}\ h_2\ xs & &
\end{array}$$

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:

$$\begin{array}{lll}
\textbf{letrec}\ h_1\ xs & = & h_2\ xs \\
\quad h_2\ xs & = & \textbf{case}\ (\ \textbf{case}\ xs\ \textbf{of} \\
 & & \qquad\quad [] \rightarrow [] \\
 & & \qquad\quad (x' : xs') \rightarrow (2 * x') : g\ xs')\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow x' + sum\ xs' \\
\textbf{in}\ h_1\ xs & &
\end{array}$$

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

$$\begin{array}{lll}
\textbf{letrec}\ h_1\ xs & = & h_2\ xs \\
\quad h_2\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow 2 * x' + sum\ (g\ xs') \\
\textbf{in}\ h_1\ xs & &
\end{array}$$

We notice that unlike in previous examples, $sum\ (g\ xs')$ is not similar to what we started transforming and we can therefore continue the transformation. We inline the body of sum, perform the substitution of its arguments and inline the body of g:

$$\begin{array}{lll}
\textbf{letrec}\ h_1\ xs & = & h_2\ xs \\
\quad h_2\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [] \rightarrow 0 \\
 & & \quad (x' : xs') \rightarrow 2 * x' + h_3\ xs' \\
\quad h_3\ xs' & = & h_4\ xs'
\end{array}$$

$$\begin{array}{l}
\quad\quad h_4\ xs' \ = \ \textbf{case}\ (\ \textbf{case}\ xs'\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad [] \ \rightarrow\ [] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (x'' : xs'') \ \rightarrow\ (3 * x'') : f\ xs'')\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad [] \ \rightarrow\ 0 \\
\quad\quad\quad\quad\quad\quad (x' : xs') \ \rightarrow\ x' \ +\ sum\ xs' \\
\textbf{in}\ h_1\ xs
\end{array}$$

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

$$\begin{array}{l}
\textbf{letrec}\ h_1\ xs \ = \ h_2\ xs \\
\quad\quad h_2\ xs \ = \ \textbf{case}\ xs\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad [] \ \rightarrow\ 0 \\
\quad\quad\quad\quad\quad\quad (x' : xs') \ \rightarrow\ 2 * x' \ +\ h_3\ xs' \\
\quad\quad h_3\ xs' \ = \ h_4\ xs' \\
\quad\quad h_4\ xs' \ = \ \textbf{case}\ xs'\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad [] \ \rightarrow\ 0 \\
\quad\quad\quad\quad\quad\quad (x'' : xs'') \ \rightarrow\ 3 * x'' \ +\ sum\ (f\ xs'') \\
\textbf{in}\ h_1\ xs
\end{array}$$

We notice a familiar expression in $sum\ (f\ xs'')$, and fold against h_1:

$$\begin{array}{l}
\textbf{letrec}\ h_1\ xs \ = \ h_2\ xs \\
\quad\quad h_2\ xs \ = \ \textbf{case}\ xs\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad [] \ \rightarrow\ 0 \\
\quad\quad\quad\quad\quad\quad (x' : xs') \ \rightarrow\ 2 * x' \ +\ h_3\ xs' \\
\quad\quad h_3\ xs' \ = \ h_4\ xs' \\
\quad\quad h_4\ xs' \ = \ \textbf{case}\ xs'\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad [] \ \rightarrow\ 0 \\
\quad\quad\quad\quad\quad\quad (x'' : xs'') \ \rightarrow\ 3 * x'' \ +\ h_1\ xs'' \\
\textbf{in}\ h_1\ xs
\end{array}$$

The new functions h_2 and h_4 both consume a list and returns a number, so our algorithm has eliminated the intermediate list between f and sum.

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

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

This is simplified by our positive supercompiler to:

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

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 $|xs| = |ys|$).

Figure 2.11 contains the necessary change to rule R9 for our supercompilation algorithm can also be used for languages with call-by-value semantics.

$$\begin{array}{lll} \mathcal{D}[\![\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2]\!]\,\rho\,\mathcal{R} & \mid x \in \mathit{linear}(e_2) \text{ and } x \in \mathit{strict}(e_2) & = \mathcal{D}[\![[e_1/x]e_2]\!]\,\rho\,\mathcal{R} \\ & \mid \text{otherwise} & = \mathbf{let}\,\mathcal{D}[\![e_1]\!]\,\rho\,\epsilon\,\mathbf{in}\,\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R} \end{array} \quad \text{(R9)}$$

Figure 2.11: Call-by-value adjustment for rule R9

CHAPTER 3

Algorithm Correctness

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

For proving both termination and total correctness we need a general context *C* which has zero or more holes in the place of some subexpressions. We return to using the more conventional notation of $C[e]$ to replace all holes in *C* with *e*.

3.1 Termination

In order to prove that the algorithm terminates we show that each recursive application of $\mathcal{D}[\![]\!]\ \rho$ and $\mathcal{B}[\![]\!]\ \rho$ in the right-hand sides of Figure 2.6 and 2.7 has a strictly smaller weight than the left-hand side.

The weight of an arithmetic expression is two plus the sum of the weight of its subexpressions and the weight of all other expressions is one plus the sum of the weight of their subexpressions.

Definition 3.1. *The weight of an expression is* $|s(e_1,\ldots,e_n)| = 1 + \sum_{i=1}^{n} |e_i|$.

We also define the weight of a context $\mathcal{R}$ as the sum of the weight of its frames, and define the weight of each frame as a constant plus the weight of the expressions in the frame.

Definition 3.2. *The weight of a context* $\mathcal{R} = \mathcal{F}_1 : \ldots : \mathcal{F}_n$ *is the sum of the weight of its frames:* $|\mathcal{R}| = \sum_{i=1}^{n} |F_i|$

Definition 3.3. *The weight of a frame is:*

$$\begin{array}{rcl}
|\,\square\ e| &=& 1+|e| \\
|\,\square \oplus e| &=& 1+|e| \\
|n \oplus \square| &=& 1+|n| \\
|\textbf{case}\ \square\ \textbf{of}\,\{p_i \to e_i\}| &=& 1+\sum |e_i|
\end{array}$$

Having established the weights of expressions and contexts, we need to introduce some technical machinery to be used for the relation $\trianglelefteq$.

Definition 3.4. *Let S be a set with a relation $\leq$. Then $(S, \leq)$ is a quasi-order if $\leq$ is reflexive and transitive.*

Definition 3.5. *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 3.6 (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 3.7 (Kruskal's Tree Theorem)**.** *If S is a finite set of function symbols, then any infinite sequence $t_1, t_2, \ldots$ of terms from the set S contains two terms t_i and t_j with $i < j$ such that $t_i \trianglelefteq t_j$.*

Proof (Similar to Dershowitz (1987)). Collapse all integers to a single 0-ary constructor, and all variables to a different 0-ary constructor.

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

Since the set *S* of function symbols is well-quasi-ordered by $\geq$, there must exist an infinite subsequence $\bar{r}$ of $\bar{t}$, 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 3.6 that the set of finite sequences consisting of the immediate subterms of the elements in $\bar{r}$ is also well-quasi-ordered. But then there would have to be an embedding in $\bar{t}$ itself, in which case it would not be a counterexample. $\square$

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

Lemma 3.8. *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 function names h_i, which are not in *T*. By inspection of the rules it is clear that only rule R3 can introduce such new terms. Inspection of the RHS of these rules:

case 1: $h'\,\overline{x}$: No recursive application in the RHS.

case 2: $[\overline{\mathcal{D}[\![f]\!]\,\rho\,\epsilon}/\overline{y}]\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon$: No modification of ρ.

case 3: $\textbf{letrec}\, h = \lambda\overline{x}.\mathcal{D}[\![v]\!]\,\rho'\,\mathcal{R}\,\textbf{in}\, h\,\overline{x}$: 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*.

$\square$

We know that there are only terms from the set *T* in the memoization list by Lemma 3.8 and Theorem 3.7 gives us that $\trianglelefteq$ is a well-quasi-order.

Corollary 3.9. *For all memoization lists $(h, \lambda\overline{x}.[g]\mathcal{R}) : \rho$ produced by the supercompiler, there exists no $(h', \lambda\overline{x}'.[g]\mathcal{R}') \in \rho$ such that $[g]\mathcal{R}' \trianglelefteq [g]\mathcal{R}$.*

We define an ordering of memoization lists, where a longer list is considered smaller:

Definition 3.10. *The order between two memoization lists ρ and ρ' is $\rho' < \rho$ if ρ is a suffix of ρ'.*

This order is well-founded since the memoization list is never extended by terms not in *T* by Lemma 3.8 and the relation $\trianglelefteq$ is a well-quasi-order over *T* according to Lemma 3.7.

Lemma 3.11. *The order $<$ between memoization lists is well-founded.*

We need two new constants in the weight, the first constant *M* is the number of case expressions in an expression, and the second constant *A* is the number of arithmetic expressions and left frames in an expression It will be convenient to allow these expressions to contain holes.

Definition 3.12 (Left frame). *A left frame $\mathcal{F}$ is a frame on the form $\square \oplus e$.*

Definition 3.13. *The constant M_e is the number of case-expressions in the expression e.*

Definition 3.14. *The constant $A_{[e]\mathcal{R}}$ is the sum of the number of arithmetic expressions in e and the number of left frames in $\mathcal{R}$.*

We define the weight of driving and building an expression as a combination of the memoization list, the number of case expressions in the expression to be transformed, the size of the expression to be transformed, the number of arithmetic expressions and left frames, and the size of the expression currently in focus:

Definition 3.15. *The weight of a driving problem* $|\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}| = (\rho, M_{[e]\mathcal{R}}, |[e]\mathcal{R}|, A_{[e]\mathcal{R}}, |e|)$ *and the weight of a building problem* $|\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}| = (\rho, M_{\mathcal{R}}, |\mathcal{R}|, A_{\mathcal{R}}, 0)$

Tuples must be ordered for us to tell whether the weight of the transformation actually decreases. We define the weight to be smaller for tuples with long ρ's, and use the standard lexical order between tuples for the last three components:

Definition 3.16. *The order between two tuples* $(\rho, n_1, n_2, n_3, n_4)$ *and* $(\rho', m_1, m_2, m_3, m_4)$ *is:*

$$\begin{array}{ll}
(\rho', n_1, n_2, n_3, n_4) < (\rho, m_1, m_2, m_3, m_4) & \textit{if } \rho' < \rho \\
(\rho, n_1, n_2, n_3, n_4) < (\rho, m_1, m_2, m_3, m_4) & \textit{if } n_1 < m_1 \\
(\rho, n_1, n_2, n_3, n_4) < (\rho, n_1, m_2, m_3, m_4) & \textit{if } n_2 < m_2 \\
(\rho, n_1, n_2, n_3, n_4) < (\rho, n_1, n_2, m_3, m_4) & \textit{if } n_3 < m_3 \\
(\rho, n_1, n_2, n_3, n_4) < (\rho, n_1, n_2, n_3, m_4) & \textit{if } n_4 < m_4
\end{array}$$

The order between tuples is well founded since the relations between the components are well founded:

Lemma 3.17. *The relation $<$ on tuples is well-founded.*

We can now formulate a lemma and prove that the weight is decreasing for each transformation step of our algorithm. The full proof is included in Appendix B.1.

Lemma 3.18. *For each rule* $\mathcal{D}[\![e_1]\!]\,\rho\,\mathcal{R}_1 = e_1'$ *and* $\mathcal{B}[\![e_2]\!]\,\rho\,\mathcal{R}_2 = e_2'$ *in Definition 2.6 and 2.7 and each call* $\mathcal{D}[\![e_3]\!]\,\rho'\,\mathcal{R}_3$ *or* $\mathcal{B}[\![e_4]\!]\,\rho\,\mathcal{R}_4$ *in* e_1' *or* e_2', $|\mathcal{D}[\![e_3]\!]\,\rho'\,\mathcal{R}_3| < |\mathcal{D}[\![e_1]\!]\,\rho\,\mathcal{R}_1|$, $|\mathcal{B}[\![e_4]\!]\,\rho\,\mathcal{R}_4| < |\mathcal{D}[\![e_1]\!]\,\rho\,\mathcal{R}_1|$, $|\mathcal{D}[\![e_3]\!]\,\rho'\,\mathcal{R}_3| < |\mathcal{B}[\![e_2]\!]\,\rho\,\mathcal{R}_2|$, *and* $|\mathcal{B}[\![e_4]\!]\,\rho\,\mathcal{R}_4| < |\mathcal{B}[\![e_2]\!]\,\rho\,\mathcal{R}_2|$.

We need a lemma about totality of the driving and building algorithms since they might otherwise be undefined for some inputs.

Lemma 3.19 (Totality)**.** *For all expressions* $[e]\mathcal{R}$, $\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}$ *and* $\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}$ *are matched by a unique rule in Figure 2.6.*

Proof. The driving algorithm is total because rule R14 will cover all expressions that do not match any of the previous rules. The building algorithm is total because rules R15-R17 and R19-R20 match on all possible frame types of $\mathcal{R}$. □

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

Theorem 3.20 (Termination)**.** *The driving algorithm* $\mathcal{D}[\![]\!]\,\rho$ *and building algorithm* $\mathcal{B}[\![]\!]\,\rho$ *terminates for all inputs.*

Proof. We know that the weight of the transformation decreases for each step by Lemma 3.18 and we also know that each recursive application will match a rule by Lemma 3.19. Since the ordering $<$ over tuples is well founded according to Lemma 3.17 the system will eventually terminate. □

3.2 Total Correctness

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

Sands (1996a) shows how a transformation can change the semantics in rather subtle ways – consider the function:

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

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

Definition 3.21 (Operational Approximation and Equivalence)**.**

- *e operationally approximates e', $e \sqsubseteq e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed, if evaluation of $C[e]$ terminates then so does evaluation of $C[e']$.*
- *e is operationally equivalent to e', $e \cong e'$, 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 3.22 (Improvement, Strong Improvement)**.**

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

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

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

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

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

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

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

Theorem 3.23 (Sands Improvement theorem). *If* $g \overset{def}{=} e$ *and* $e \trianglerighteq C[g]$ *then* $g \trianglerighteq h$ *where* $h = C[h]$.

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

Theorem 3.25 (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 3.26 (Partial Correctness). *If* $e_i \cong e'_i$ *for all* $i \in I$ *then* $h_i \precsim g_i$*,* $i \in I$.

which we combine with Theorem 3.23 to get total correctness for a transformation:

Corollary 3.27. *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 3.28 (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* $e \precsim e'$.

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

Lemma 3.29 (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 in order to prove total correctness. Since strong improvement is improvement in one direction and operational approximation in the other direction, a set of approximation laws that correspond to the improvement laws in Lemma 3.28 is necessary.

Lemma 3.30. *Approximation laws*

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

Combining Lemma 3.28 and Lemma 3.30 gives us the final tools we need to prove strong improvement:

Lemma 3.31. *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 recursive definitions which the improvement theorem is defined for. This is solved by a technical lemma that relates the cost of terms on a certain form to their recursive counterparts.

Theorem 3.32. *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 3.29 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 3.25, 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 for proving our local improvement theorem

Theorem 3.33. *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 3.34 (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 Theorem 3.32 $g \trianglelefteq\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_0)$. Use this, the congruence properties, and the properties listed in Theorem 3.33 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 3.27, $g \trianglerighteq_s g'$ 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 Theorem 3.32, $g' \trianglelefteq\trianglerighteq fix\,(\lambda h.\lambda\overline{x}.e_1)$. 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 Theorem 3.33 we get $\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, h \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, h$, which can be further expanded by congruency properties of improvement to $[\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, h/h]e \trianglerighteq_s [\textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, h/h]e$. Using Theorem 3.33 one more time yields $\textbf{letrec}\, h = \lambda\overline{x}.e_0 \,\textbf{in}\, e \trianglerighteq_s \textbf{letrec}\, h = \lambda\overline{x}.e_1 \,\textbf{in}\, e$ which proves our theorem. □

Lemma 3.35. *Let $[e]\mathcal{R}$ be an expression and ρ a memoization list such that*

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

then $[e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R})$ if $[e']\mathcal{R}' \trianglerighteq_s \rho'(\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}')$ for all $e', \mathcal{R}', \rho'$ such that $|\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}'| < |\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}|$.

The full proof is included in Appendix B.3.

Theorem 3.36 (Total Correctness)**.** *Let $[e]\mathcal{R}$ be an expression and ρ a memoization list such that*

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

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

The full proof is included in Appendix B.4, but the proof does not reveal anything unexpected. The proof for rule R3 is similar in structure to the proof by Sands (1996a, p. 24).

CHAPTER 4

Controlling and Extending Supercompilation

We have so far defined a supercompilation algorithm in Chapter 2 and proved it correct in Chapter 3. If we implement this algorithm in a compiler and try to optimize real programs it will turn out that we often suffer from code explosion: the resulting binary can be several times larger than the binary of the same program that is not supercompiled. Even if there is sufficient storage space available there are still downsides of large binaries, the most obvious one being compilation times: sometimes even small programs take minutes rather than seconds to compile.

If we add some pre- and post-conditions to certain rules in the algorithm, and replace local function definitions in rule R3 with shared top level functions, these problems can be mitigated. This chapter describes the useful conditions we have found, and the problems that they are meant to solve. These conditions should be straightforward to transfer to other more powerful supercompilers.

These adjustments gives us a faster algorithm, which in turn allows us to control code explosion by speculatively supercompile expressions and discard bad results. Our measurements use a simple comparison of the size of the resulting expression versus the number of reduction steps performed by the supercompiler.

Our guiding principle has been simple: sacrifice optimization opportunities if it significantly improves the performance or simplifies the implementation of the supercompiler. Our only concern is soundness of the supercompiler, we do not have to worry about completeness properties that could be important in theorem proving or similar domains.

There is some overlap between what problem each design choice helps to protect against.

Code Explosion The primary mechanism to tame code explosion is described in Section 4.2. Section 4.1, Section 4.3, and Section 4.4 contain design choices that help reduce the code size.

Compiler Performance Avoiding code explosion helps compilation performance, but this is not enough: Section 4.5 contains a mechanism to avoid performing work that is unlikely to give any benefit.

We have also made a lot of minor non-controversial design choices in our supercompiler which we describe in Section 4.6.

4.1 Boring Contexts

There are sometimes function calls that have no static information in their argument, making it impossible to improve them by supercompiling them in their context. By using a simple approximation of such function calls we save compilation time and avoid creating specializations that do not improve the performance. We say that such functions are in a boring context. A boring expression is an expression on the form:

$$b ::= x \mid b\,b \mid n \oplus b \mid b \oplus n \mid b \oplus b$$

A context $\mathcal{R}$ is defined as boring iff $\mathcal{R} = \epsilon$ or $\mathcal{R} = \Box\ b : \mathcal{R}'$ and $\mathcal{R}'$ is boring.

4.2 Discarding Expressions

The initial function call to reverse with an accumulating parameter, $\mathit{reverse}\ \mathit{xs}$ [], is an example that will pass the test for boring expressions since the second argument is a known constructor. Despite this there is not much the supercompiler can do with this expression and the end result will be a new function with one unfolding of reverse and a second function that is isomorphic to reverse.

Designing an algorithm that approximates what the result and how much savings were achieved from supercompiling an arbitrary expression is difficult. We can completely avoid to solve this problem by speculatively supercompile the expression, have the supercompiler keep track of the reduction steps taken, and look at the output of the supercompiler. If the number of reduction steps taken is small or the code size is deemed to large we discard the supercompiled expression and supercompile the subexpressions of the current expression instead.

The savings that are propagated by the supercompiler are used as input to a predicate *acceptable* which decides whether to discard or keep transformed functions. Our current implementation is remarkably simple:

```
$\mathit{acceptable}\ \mathit{newbinds}\ \mathit{before}\ \mathit{after}\ \mathit{savings}\ =$
 $\mathit{sizeof}(\mathit{before})\ >\ \mathit{sizeof}(\mathit{after})\ +\ \mathit{sizeof}(\mathit{newbinds})\ ||$
 $\mathit{savings}\ >=\ 5\ \&\&$
      $(\mathit{sizeof}(\mathit{after})\ +\ \mathit{sizeof}(\mathit{newbinds}))\ `\mathit{div}`\ \mathit{sizeof}(\mathit{before})\ <\ \mathit{savings}\ `\mathit{div}`\ 3$
```

4.3 Normalization of Expressions

We perform a simple normalization of expressions before comparing them for folding possibilities. This makes the supercompiler significantly faster. The big win is the removal of identity coercions in expressions, which makes the memoization list a lot shorter.

Finding renamings of an expression is a syntactical check, and sometimes this will miss out opportunities to fold because expressions that are semantically equivalent have different forms.

Consider the two expressions $g\ x\ (1\ +\ 2)$ and $g\ y\ (2\ +\ 1)$ – it is obvious to a human that these two expressions should be foldable against each other. By performing some kind of limited normalization on these expressions the second parameter will turn out to be 3 in both cases and they can fold.

We can not hope to find a normal form for our expressions in general, but performing ordinary evaluation except inlining of function calls on all subexpressions will get a long way towards our goal. It is of course important to have a normalization that terminates.

A word of caution about making the normalization too powerful: many common optimizations change the structure of expressions. One example is changing the expression $(x + 1) + 1$ to $x + 2$ which is semantically correct and clearly saves computations, but chances are that in practice the first expression came from an expression $y\ +\ 1$ in a recursive call with an increasing parameter. The expression $(y + 1)$ will be homeomorphically embedded in both $(x\ +\ 1)\ +\ 1$ and $x\ +\ 2$, but the former will be generalized to $(z\ +\ 1,\ [(x + 1)/z])$ and which in turn will fold nicely against $y\ +\ 1$.

The normalization scheme we use is the following rewrite rules applied exhaustively on all parts of expressions:

$$\begin{array}{rcl}
n_1 \oplus n_2 & \rightsquigarrow & n, \text{ where } n = n_1 + n_2 \\
\mathbf{case}\, k_j\, \overline{e}\, \mathbf{of}\, \{p_i \to e_i\} & \rightsquigarrow & \mathbf{let}\, \overline{x_j} = \overline{e}\, \mathbf{in}\, e_j \\
\mathbf{case}\, n_j\, \mathbf{of}\, \{p_i \to e_i\} & \rightsquigarrow & e_j \\
\mathbf{let}\, x = n\, \mathbf{in}\, e & \rightsquigarrow & [n/x]e
\end{array}$$

4.4 Sharing New Function Definitions

It is quite common for the case of case-rule to fire. A typical case looks something like this:

```
case ( case $x$ of
           $p_1 \ \to\ x_1$
           ...
           $p_n \ \to\ x_n$) of
    $[] \ \to\ []$
    $(x' : xs') \ \to\ append\ xs'\ zs$
```

Once the case of case-rule is done the code is:

```
case $x$ of
  $p_1$ → case $x_1$ of
            [] → []
            $(x' : xs')$ → $append\ xs'\ zs$
  ...
  $p_n$ → case $x_n$ of
            [] → []
            $(x' : xs')$ → $append\ xs'\ zs$
```

The driving algorithm will transform the call to append multiple times, and create mulitiple new functions that are all isomorphic to append. Creating the functions at the top level and calling the same function from both branches saves both transformation work and reduces code size.

Using a state monad for the driving algorithm, and have that monad store a memoization list augmented with the function definitions will allow the second branch to just insert a call to this new function created in the first branch, thereby saving a lot of transformation effort.

However, if we fold too eagerly this will miss out on specialization opportunities and this will make the final program perform worse than necessary. We can see an example of this with $append\ (append\ xs\ ys)\ zs$, which when transformed will create a function isomorphic to append itself in the first branch. After the supercompiler has transformed the second branch it will find a recursive call to $append\ (append\ xs'\ ys)\ zs$, which can fold against both the call in the first branch, $append\ ys\ zs$, and the initial call that the transformation started from. If the former is chosen no fusion will occur.

We therefore only look for renamings, not instances, when trying to fold against things stored in the monad. If there exists a specialised version of our current function the algorithm will use that, and otherwise it will create a new specialised copy.

4.5 Tainting Functions

If the same function, regardless of context, is supercompiled in different parts of the program and the result is discarded more than a certain number of times we make the unfolding for the function unavailable. This prevents future attempts of specializing the function. The concrete mechanism is that we add a special triple on the form $(*,\ \emptyset,\ g)$ to the store each time we discard a result from supercompiling a function g in some context, and the number of such triples in the store is checked for before inlining a function.

We used being discarded 3 times as threshold for our measurements.

4.6 Miscellaneous Design Choices

When supercompiling a function call $const\ x\ y$ our supercompiler blindly emits a call to a fresh function $h\ x\ y$ whose body is defined as $(\lambda x\ y.x)\ x\ y$. It is safe to directly inline the body of $const$ at the call-site but we can increase the code sharing by allowing the supercompiler to fold against this definition from other parts of the program.

If it turns out that there is only one call to h in the entire supercompiled program the ordinary simplifier of the compiler will inline it.

The reason we use evaluation contexts in our algorithm is that there is a chance that good things can happen in the interaction between evaluation contexts and an expression in the hole. There are no such possibilities with an arbitrary context that has the hole outside the focus for evaluation.

4.7 Extended Algorithm Definition

This revised algorithm take two extra parameters compared to the algorithm presented in Chapter 2. The new parameters are a store σ and the savings from the transformations the supercompiler has done so far. New function definitions are put in the store σ, which is defined as:

$$\sigma ::= \epsilon \mid (\nu, e') : \sigma$$

The savings parameter is there to track how many reduction steps the supercompiler has performed. This is currently represented as a tree in our implementation to aid us in the design of when to discard expressions, but a production compiler should use a more efficient representation. We assign the value 1 to ordinary evaluation steps in rules R1, R2, R5, R7, R8 and R9. Rule R4, the case reduction with a known constructor, is assigned the value 2 since it both decreases code size and removes a memory allocation.

We need some more arithmetic operations on our savings beyond plain addition: # indicates that a saving was under a lambda, ? combines branches of savings such as in rule R9 and max of all savings in rules R18 and R19. The most important saving, folding, is denoted by the exclamation mark (!). These abstract savings need to be interpreted to a concrete number in order to make decisions on them. For our measurements we used regular addition of savings both under lambdas and in branches such as rule R9, and gave folding the value 4. Our design principle was that foldings should trump other savings, but this is an extremely coarse design that leaves plenty of room for future improvements.

4.8 Further Extensions

This section presents some possible variations of the basic algorithm presented in Chapter 2. We show how four orthogonal modifactions can make the supercompiler more powerful and possibly subsume other program optimizations during supercompilation.

4.8.1 Strengthening Call-by-Value Supercompilation

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

General form:

$\mathcal{D}\llbracket e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = (e', \sigma', s')$

Evaluation Rules

$$\mathcal{D}\llbracket n_j \rrbracket\, \rho\, (\textbf{case}\ \Box\ \ \textbf{of}\, \{n_i \rightarrow e_i\} : \mathcal{R})\, \sigma\, s = \mathcal{D}\llbracket e_j \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+1) \qquad \text{(R1)}$$

$$\mathcal{D}\llbracket n \rrbracket\, \rho\, (n_1 \oplus \Box : \mathcal{R})\, \sigma\, s = \mathcal{D}\llbracket n_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+1), \text{ where } n_2 = n_1 + n \qquad \text{(R2)}$$

$$\mathcal{D}\llbracket g \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s \mid g \text{ not tainted and } \mathcal{R} \text{ not boring} = \mathcal{D}_{app}(g)\, \rho\, \mathcal{R}\, \sigma\, s \qquad \text{(R3)}$$

$$\mathcal{D}\llbracket k_j \rrbracket\, \rho\, (\overline{\Box\, e} : \textbf{case}\ \Box\ \ \textbf{of}\, \{k_i\, \overline{x}_i \rightarrow e_i\} : \mathcal{R})\, \sigma\, s = \mathcal{D}\llbracket \textbf{let}\, \overline{x}_j = \overline{e}\, \textbf{in}\, e_j \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+2) \qquad \text{(R4)}$$

$$\mathcal{D}\llbracket \lambda \overline{x}.e_1 \rrbracket\, \rho\, (\overline{\Box\ e_2} : \mathcal{R})\, \sigma\, s = \mathcal{D}\llbracket \textbf{let}\, \overline{x} = \overline{e_2}\, \textbf{in}\, e_1 \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+1) \qquad \text{(R5)}$$

$$\mathcal{D}\llbracket \lambda x.e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = \texttt{let}\ (e', \sigma_1, s') = \mathcal{D}\llbracket e \rrbracket\, \rho\, \epsilon\, \sigma\, 0 \ \ \texttt{in}\, \mathcal{B}\llbracket \lambda x.e' \rrbracket\, \rho\, \mathcal{R}\, \sigma_1\, (s + \#s') \qquad \text{(R6)}$$

$$\mathcal{D}\llbracket \textbf{let}\, x = n\, \textbf{in}\, e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = \mathcal{D}\llbracket [n/x]e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+1) \qquad \text{(R7)}$$

$$\mathcal{D}\llbracket \textbf{let}\, x = y\, \textbf{in}\, e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s \mid y \text{ not fresh} = \mathcal{D}\llbracket [y/x]e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+1) \qquad \text{(R8)}$$

$$\begin{aligned}
\mathcal{D}\llbracket \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s \mid x \in \textit{linear}(e_2) &= \mathcal{D}\llbracket [e_1/x]e_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma\, (s+1) \qquad \text{(R9)}\\
\mid \text{otherwise} &= \texttt{let}\ (e'_1, \sigma_1, s_1) = \mathcal{D}\llbracket e_1 \rrbracket\, \rho\, \epsilon\, \sigma\, 0\\
&\qquad\quad (e'_2, \sigma_2, s_2) = \mathcal{D}\llbracket e_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma_1\, s\\
&\quad\ \texttt{in}\, (\textbf{let}\, x = e'_1\, \textbf{in}\, e'_2, \sigma_2, ?s_1 + s_2)
\end{aligned}$$

Focusing Rules

$$\mathcal{D}\llbracket n \rrbracket\, \rho\, (\Box \oplus e_2 : \mathcal{R})\, \sigma\, s = \mathcal{D}\llbracket e_2 \rrbracket\, \rho\, (n \oplus \Box : \mathcal{R})\, \sigma\, s \qquad \text{(R10)}$$

$$\mathcal{D}\llbracket e_1 \oplus e_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = \mathcal{D}\llbracket e_1 \rrbracket\, \rho\, (\Box \oplus e_2 : \mathcal{R})\, \sigma\, s \qquad \text{(R11)}$$

$$\mathcal{D}\llbracket e_1\, e_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = \mathcal{D}\llbracket e_1 \rrbracket\, \rho\, (\Box\ e_2 : \mathcal{R})\, \sigma\, s \qquad \text{(R12)}$$

$$\mathcal{D}\llbracket \textbf{case}\, e\ \textbf{of}\, \{p_i \rightarrow e_i\} \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = \mathcal{D}\llbracket e \rrbracket\, \rho\, (\textbf{case}\ \Box\ \ \textbf{of}\, \{p_i \rightarrow e_i\} : \mathcal{R})\, \sigma\, s \qquad \text{(R13)}$$

Fallthrough

$$\mathcal{D}\llbracket e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s = \mathcal{B}\llbracket e \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s \qquad \text{(R14)}$$

Rebuilding Expressions

$$\mathcal{B}\llbracket e' \rrbracket\, \rho\, (\Box \oplus e_2 : \mathcal{R})\, \sigma\, s = \texttt{let}\ (e'_2, \sigma_1, s') = \mathcal{D}\llbracket e_2 \rrbracket\, \rho\, \epsilon\, \sigma\, s \ \ \texttt{in}\, \mathcal{B}\llbracket e' \oplus e'_2 \rrbracket\, \rho\, \mathcal{R}\, \sigma_1\, s' \qquad \text{(R15)}$$

$$\mathcal{B}\llbracket e' \rrbracket\, \rho\, (e'_1 \oplus \Box : \mathcal{R})\, \sigma\, s = \mathcal{B}\llbracket e'_1 \oplus e' \rrbracket\, \rho\, \mathcal{R}\, \sigma\, s \qquad \text{(R16)}$$

$$\mathcal{B}\llbracket e' \rrbracket\, \rho\, (\Box\, e : \mathcal{R})\, \sigma\, s = \texttt{let}\ (e'', \sigma_1, s') = \mathcal{D}\llbracket e \rrbracket\, \rho\, \epsilon\, \sigma\, s \ \ \texttt{in}\, \mathcal{B}\llbracket e' e'' \rrbracket\, \rho\, \mathcal{R} s' \qquad \text{(R17)}$$

$$\begin{aligned}
\mathcal{B}\llbracket x' \rrbracket\, \rho\, (\textbf{case}\ \Box\ \ \textbf{of}\, \{p_i \rightarrow e_i\} : \mathcal{R})\, \sigma\, s &= \texttt{let}\ (e'_i, \sigma_i, s_i) = \mathcal{D}\llbracket [p_i/x]e_i \rrbracket\, \rho\, ([p_i/x]\mathcal{R})\, \sigma_{i-1}\, s \qquad \text{(R18)}\\
&\quad\ \texttt{in}\, (\textbf{case}\, x'\ \textbf{of}\, \{p_i \rightarrow e'_i\}, \sigma_i, max(s_i))
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}\llbracket e' \rrbracket\, \rho\, (\textbf{case}\ \Box\ \ \textbf{of}\, \{p_i \rightarrow e_i\} : \mathcal{R})\, \sigma\, s &= \texttt{let}\ (e'_i, \sigma_i, s_i) = \mathcal{D}\llbracket e_i \rrbracket\, \rho\, \mathcal{R}\, \sigma_{i-1}\, s \qquad \text{(R19)}\\
&\quad\ \texttt{in}\, (\textbf{case}\, e'\ \textbf{of}\, \{p_i \rightarrow e'_i\}, \sigma_i, max(s_i))
\end{aligned}$$

$$\mathcal{B}\llbracket e' \rrbracket\, \rho\, \epsilon\, \sigma\, s = (e', \sigma, s) \qquad \text{(R20)}$$

Figure 4.1: Driving algorithm and building algorithm

$$
\begin{array}{lll}
\mathcal{D}_{app}(g)\,\rho\,\mathcal{R}\,\sigma\,s \mid \exists h.(\sigma|_g \cup \rho|_g)(h) \equiv \lambda\overline{x}.e_2 & = (h\,\overline{x}, \sigma, !s) & (1)\\
\qquad\qquad\quad \mid \exists h.\rho|_g(h) \trianglelefteq \lambda\overline{x}.e_2 & = \texttt{let}\,(f_g, [\overline{f}/\overline{y}]) = divide(\lambda\overline{x}.e_2, \rho(h)) & (2)\\
 & \qquad (\overline{f}', \sigma_i, s_i) = \mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon\,\sigma_{i-1}\,s_{i-1} & \\
 & \qquad (f_g', \sigma_2, s_2) = \mathcal{D}[\![f_g]\!]\,\rho\,\epsilon\,\sigma_i\,s_i & \\
 & \quad \texttt{in}\,([\overline{f}'/\overline{y}]f_g', \sigma_2, s_i + s_2)) & \\
\qquad\qquad\quad \mid \textit{acceptable}(\sigma_1 \backslash \sigma, [g]\mathcal{R}, e', s_1) & = (h\,\overline{x}, ((h, \lambda\overline{x}.e_2), e') : \sigma_1, s_1) & (3)\\
\qquad\qquad\quad \mid \text{otherwise} & = \mathcal{B}[\![g]\!]\,\rho\,\mathcal{R}\,((*, \emptyset, g) : \sigma)\,s & (4)\\
\quad \text{where } (e', \sigma_1, s_1) = \mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R}\,\sigma s & & \\
\qquad g \overset{\text{def}}{=} e & & \\
\qquad e_2 = \textit{normalize}([g]\mathcal{R}) & & \\
\qquad \rho' = (h, \lambda\overline{x}.e_2) : \rho & & \\
\qquad h \text{ fresh} & & \\
\qquad \overline{x} = \textit{fv}(e_2) & &
\end{array}
$$

Figure 4.2: Driving of applications

both of these requirements, or functions that are complex enough to fool the analyses used by the supercompiler. We already mentioned one example of this problem in in section 2.3: $zip\ (map\ f\ xs)\ (map\ g\ ys)$.

If the supercompiler instead propagates let-expressions into the branches of case-expressions it simplifies the job for the analyses since they no longer need to account for different behaviours in different branches. Not only does this modification increase the precision of the analyses, but it also allows our improved supercompiler to remove more constructions that cause memory allocations. The propagation of let-expressions is orthogonal to the amount of information propagated, so it works for both positive (Sørensen et al. 1996) and perfect supercompilation (Glück and Klimov 1993). We illustrate the increased strength through the following example:

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

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

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

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

Shortcut deforestation (Gill 1996) removes one of the intermediate lists in the first example, but does not remove arbitrary algebraic data types.

Stream fusion (Coutts et al. 2007) removes both the intermediate lists in the first example, but does not remove arbitrary algebraic data types without manual extensions.

Positive supercompilation (Sørensen et al. 1996) for a strict language removes the first intermediate structure in both examples, and for a lazy language it removes both lists and both trees.

We present one more step towards allowing the programmer to write clear and concise code in strict languages while getting good performance. and we start out with a step by step example where our improved supercompiler removes both intermediate lists for *zip* to give the reader an intuitive feel for how the algorithm behaves. Our first example is transformation of the standard function *zip*, which takes two lists as parameters: $zip\ (map\ f\ xs')\ (map\ g\ ys')$.

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

```
letrec h0 f xs' g ys' = h1 f xs' g ys'
       h1 f xs' g ys' = let ys = map g ys'
                        in case map f xs' of
                           [] → []
                           (x' : xs') → case ys of
                                          [] → []
                                          (y' : ys') → (x', y') : zip xs' ys'
in h0 f xs' g ys'
```

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

```
letrec h0 f xs' g ys' = h1 f xs' g ys'
       h1 f xs' g ys' = h2 f xs' g ys'
       h2 f xs' g ys' = let ys = map g ys'
                        in case ( case xs' of
                                    [] → []
                                    (z : zs) → f z : map f zs) of
                             [] → []
                             (x' : xs') → case ys of
                                            [] → []
                                            (y' : ys') → (x', y') : zip xs' ys'
in h0 f xs' g ys'
```

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

```
letrec h0 f xs' g ys' = h1 f xs' g ys'
       h1 f xs' g ys' = h2 f xs' g ys'
```

```
        h2 f xs' g ys' = case xs' of
                           [] → let ys = map g ys'
                                in case [] of
                                     [] → []
                                     (x' : xs') → case ys of
                                                    [] → []
                                                    (y' : ys') → (x', y') : zip xs' ys'
                           (z : zs) → let ys = map g ys'
                                      in case f z : map f zs of
                                           [] → []
                                           (x' : xs') → case ys of
                                                          [] → []
                                                          (y' : ys') → (x', y') : zip xs' ys'
in h0 f xs' g ys'
```

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

```
letrec h0 f xs' g ys' = h1 f xs' g ys'
       h1 f xs' g ys' = h2 f xs' g ys'
       h2 f xs' g ys' = case xs' of
                          [] → let ys = map g ys' in []
                          (z : zs) → let ys = map g ys'
                                     in let x' = f z, xs' = map f zs
                                     in case ys of
                                          [] → []
                                          (y' : ys') → (x', y') : zip xs' ys'
in h0 f xs' g ys'
```

The first branch can either be left as is, or one can transform the let-expression to get a new function that is isomorphic to *map* and a call to it. This is an orthogonal problem to removing multiple intermediate structures however, and we will not treat it further in this example. In Section 4.8.1.2 we show how to automatically remove superfluous let-expressions such as this through termination analysis. The reduction of the case-expression in the second branch reveals that the second branch is strict in *ys*, so *ys* will be evaluated, and the termination behavior will be the same even after performing the substitution. After performing the substitution we have:

```
letrec h0 f xs' g ys' = h1 f xs' g ys'
       h1 f xs' g ys' = h2 f xs' g ys'
```

$$\begin{array}{lrcl}
& h_2\; f\; xs'\; g\; ys' & = & \mathbf{case}\; xs'\; \mathbf{of} \\
& & & \quad [] \;\rightarrow\; \mathbf{let}\; ys \;=\; map\; g\; ys'\; \mathbf{in}\; [] \\
& & & \quad (z : zs) \;\rightarrow\; \mathbf{let}\; x' \;=\; f\; z,\; xs' \;=\; map\; f\; zs \\
& & & \qquad\qquad\qquad \mathbf{in\; case}\; map\; g\; ys'\; \mathbf{of} \\
& & & \qquad\qquad\qquad\quad [] \;\rightarrow\; [] \\
& & & \qquad\qquad\qquad\quad (y' : ys') \;\rightarrow\; (x', y') : zip\; xs'\; ys' \\
\mathbf{in}\; h_0\; f\; xs'\; g\; ys' & & &
\end{array}$$

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

$$\begin{array}{lrcl}
\mathbf{letrec} & h_0\; f\; xs'\; g\; ys' & = & h_1\; f\; xs'\; g\; ys' \\
& h_1\; f\; xs'\; g\; ys' & = & h_2\; f\; xs'\; g\; ys' \\
& h_2\; f\; xs'\; g\; ys' & = & \mathbf{case}\; xs'\; \mathbf{of} \\
& & & \quad [] \;\rightarrow\; \mathbf{let}\; ys \;=\; map\; g\; ys'\; \mathbf{in}\; [] \\
& & & \quad (z : zs) \;\rightarrow\; h_3\; f\; z\; zs\; ys'\; g \\
& h_3\; f\; z\; zs\; ys'\; g & = & \mathbf{let}\; x' \;=\; f\; z,\; xs' \;=\; map\; f\; zs \\
& & & \mathbf{in\; case}\; (\; \mathbf{case}\; ys'\; \mathbf{of} \\
& & & \qquad\qquad\qquad [] \;\rightarrow\; [] \\
& & & \qquad\qquad\qquad (z' : zs') \;\rightarrow\; g\; z' : map\; g\; zs')\; \mathbf{of} \\
& & & \qquad\quad [] \;\rightarrow\; [] \\
& & & \qquad\quad (y' : ys') \;\rightarrow\; (x', y') : zip\; xs'\; ys' \\
\mathbf{in}\; h_0\; f\; xs'\; g\; ys' & & &
\end{array}$$

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

$$\begin{array}{lrcl}
\mathbf{letrec} & h_0\; f\; xs'\; g\; ys' & = & h_1\; f\; xs'\; g\; ys' \\
& h_1\; f\; xs'\; g\; ys' & = & h_2\; f\; xs'\; g\; ys' \\
& h_2\; f\; xs'\; g\; ys' & = & \mathbf{case}\; xs'\; \mathbf{of} \\
& & & \quad [] \;\rightarrow\; \mathbf{let}\; ys \;=\; map\; g\; ys'\; \mathbf{in}\; [] \\
& & & \quad (z : zs) \;\rightarrow\; h_3\; f\; z\; zs\; ys'\; g \\
& h_3\; f\; z\; zs\; ys'\; g & = & \mathbf{case}\; ys'\; \mathbf{of} \\
& & & \quad [] \;\rightarrow\; \mathbf{let}\; x' \;=\; f\; z,\; xs' \;=\; map\; f\; zs \\
& & & \qquad\quad \mathbf{in\; case}\; []\; \mathbf{of} \\
& & & \qquad\qquad [] \;\rightarrow\; [] \\
& & & \qquad\qquad (y' : ys') \;\rightarrow\; (x', y') : zip\; xs'\; ys' \\
& & & \quad (z' : zs') \;\rightarrow\; \mathbf{let}\; x' \;=\; f\; z,\; xs' \;=\; map\; f\; zs \\
& & & \qquad\qquad\qquad \mathbf{in\; case}\; g\; z' : map\; g\; zs'\; \mathbf{of} \\
& & & \qquad\qquad\qquad\quad [] \;\rightarrow\; [] \\
& & & \qquad\qquad\qquad\quad (y' : ys') \;\rightarrow\; (x', y') : zip\; xs'\; ys' \\
\mathbf{in}\; h_0\; f\; xs'\; g\; ys' & & &
\end{array}$$

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

$$
\begin{array}{l}
\textbf{letrec}\ h_0\ f\ xs'\ g\ ys' \ = \ h_1\ f\ xs'\ g\ ys' \\
\qquad h_1\ f\ xs'\ g\ ys' \ = \ h_2\ f\ xs'\ g\ ys' \\
\qquad h_2\ f\ xs'\ g\ ys' \ = \ \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad [] \rightarrow \textbf{let}\ ys = map\ g\ ys'\ \textbf{in}\ [] \\
\qquad\qquad (z : zs) \rightarrow h_3\ f\ z\ zs\ ys'\ g \\
\qquad h_3\ f\ z\ zs\ ys'\ g \ = \ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad [] \rightarrow \textbf{let}\ x' = f\ z,\ xs' = map\ f\ zs\ \textbf{in}\ [] \\
\qquad\qquad (z' : zs') \rightarrow \textbf{let}\ x' = f\ z,\ xs' = map\ f\ zs \\
\qquad\qquad\qquad \textbf{in}\ (x', g\ z') : zip\ xs'\ (map\ g\ zs') \\
\textbf{in}\ h_0\ f\ xs'\ g\ ys'
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{letrec}\ h_0\ f\ xs'\ g\ ys' \ = \ h_1\ f\ xs'\ g\ ys' \\
\qquad h_1\ f\ xs'\ g\ ys' \ = \ h_2\ f\ xs'\ g\ ys' \\
\qquad h_2\ f\ xs'\ g\ ys' \ = \ \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad [] \rightarrow \textbf{let}\ ys = map\ g\ ys'\ \textbf{in}\ [] \\
\qquad\qquad (z : zs) \rightarrow h_3\ f\ z\ zs\ ys'\ g \\
\qquad h_3\ f\ z\ zs\ ys'\ g \ = \ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad [] \rightarrow \textbf{let}\ x' = f\ z,\ xs' = map\ f\ zs\ \textbf{in}\ [] \\
\qquad\qquad (z' : zs') \rightarrow (f\ z, g\ z') : zip\ (map\ f\ zs)\ (map\ g\ zs') \\
\textbf{in}\ h_0\ f\ xs'\ g\ ys'
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{letrec}\ h_0\ f\ xs'\ g\ ys' \ = \ h_1\ f\ xs'\ g\ ys' \\
\qquad h_1\ f\ xs'\ g\ ys' \ = \ h_2\ f\ xs'\ g\ ys' \\
\qquad h_2\ f\ xs'\ g\ ys' \ = \ \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad [] \rightarrow \textbf{let}\ ys = map\ g\ ys'\ \textbf{in}\ [] \\
\qquad\qquad (z : zs) \rightarrow h_3\ f\ z\ zs\ ys'\ g \\
\qquad h_3\ f\ z\ zs\ ys'\ g \ = \ \textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad [] \rightarrow \textbf{let}\ x' = f\ z,\ xs' = map\ f\ zs\ \textbf{in}\ [] \\
\qquad\qquad (z' : zs') \rightarrow (f\ z, g\ z') : h_0\ f\ zs\ g\ zs' \\
\textbf{in}\ h_0\ f\ xs'\ g\ ys'
\end{array}
$$

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

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

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

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

We transform the expression $zipT\ (mapT\ f\ xs)\ (mapT\ g\ ys)$, which applies f to the first tree, g to the second tree and create a final tree whose nodes consists of pairs of the data from the two intermediate trees. We start our transformation by allocating a new fresh function name (h_0) and repeat the transformation steps that we just saw for the list case. The end result is:

```
letrec h₀ f xs g ys = h₁ f xs g ys
       h₁ f xs g ys = h₂ f xs g ys
       h₂ f xs g ys = case xs of
                        Empty → let ys' = mapT g ys in Empty
                        Node l a r → h₃ g ys f l a r
       h₃ g ys f l a r = case ys of
                           Empty → let l₁ = mapT f l, a₁ = f a, r₁ = mapT f r
                                   in Empty
                           Node l' a' r' → Node (h₀ f l g l') (f a, g a') (h₀ f r g r')
in h₀ f xs g ys
```

The same result as in the list case: the new function h_0 does not pass any intermediate trees for the common case: when both *xs* and *ys* are non-empty. If one of them is empty, it is necessary to run *mapT* on the remaining part of the other tree. This example also highlights the need to discard unused let-bindings.

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

let $x = e$ **in** $fst\ (x,\ x)$

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

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

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

Evaluation Rules

$$\begin{array}{lcll}
\mathcal{D}\llbracket n_j \rrbracket\,\Gamma\,\rho\,(\textbf{case}\ \Box\ \ \textbf{of}\,\{n_i \to e_i\} : \mathcal{R}) & = & \mathcal{D}\llbracket \textbf{let}\,\Gamma\,\textbf{in}\,[e_j]\mathcal{R} \rrbracket\,\emptyset\,\rho\,\epsilon & \text{(R1)}\\
\mathcal{D}\llbracket n \rrbracket\,\Gamma\,\rho\,(n_1 \oplus \Box : \mathcal{R}) & = & \mathcal{D}\llbracket n_2 \rrbracket\,\Gamma\,\rho\,\mathcal{R},\ \text{where}\ n_2 = n_1 + n & \text{(R2)}\\
 & & & \\
\mathcal{D}\llbracket g \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{D}_{app}(g)\,\Gamma\,\rho\,\mathcal{R} & \text{(R3)}\\
 & & & \\
\mathcal{D}\llbracket k_j \rrbracket\,\Gamma\,\rho\,(\overline{\Box\, e} : \textbf{case}\ \Box\ \ \textbf{of}\,\{k_i\,\overline{x}_i \to e_i\} : \mathcal{R}) & = & \mathcal{D}\llbracket \textbf{let}\,\Gamma\,\textbf{in}\,[\textbf{let}\,\overline{x}_j = \overline{e}\,\textbf{in}\,e_j]\mathcal{R} \rrbracket\,\emptyset\,\rho\,\epsilon & \text{(R4)}\\
 & & & \\
\mathcal{D}\llbracket \lambda\overline{x}.e_1 \rrbracket\,\Gamma\,\rho\,(\overline{\Box\ e_2} : \mathcal{R}) & = & \mathcal{D}\llbracket \textbf{let}\,\Gamma\,\textbf{in}\,[\textbf{let}\,\overline{x} = \overline{e_2}\,\textbf{in}\,e_1]\mathcal{R} \rrbracket\,\emptyset\,\rho\,\epsilon & \text{(R5)}\\
\mathcal{D}\llbracket \lambda x.e \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{B}\llbracket \lambda x.(\mathcal{D}\llbracket e \rrbracket\,\emptyset\,\rho\,\epsilon) \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R6)}\\
 & & & \\
\mathcal{D}\llbracket \textbf{let}\,x = n\,\textbf{in}\,e \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{D}\llbracket [n/x]e \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R7)}\\
\mathcal{D}\llbracket \textbf{let}\,x = y\,\textbf{in}\,e \rrbracket\,\Gamma\,\rho\,\mathcal{R} \mid y\ \text{not freshly generated} & = & \mathcal{D}\llbracket [y/x]e \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R8)}\\
\mathcal{D}\llbracket \textbf{let}\,x = e_1\,\textbf{in}\,e_2 \rrbracket\,\Gamma\,\rho\,\mathcal{R} \mid x \in \mathit{linear}(e_2) & & & \text{(R9)}\\
\qquad\qquad\qquad\qquad ,\ x \in \mathit{strict}(e_2) & = & \mathcal{D}\llbracket [e_1/x]e_2 \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \\
\qquad\qquad\qquad\qquad \mid \text{otherwise} & = & \mathcal{D}\llbracket e_2 \rrbracket\,(\Gamma; x = e_1)\,\rho\,\mathcal{R} &
\end{array}$$

Focusing Rules

$$\begin{array}{lcll}
\mathcal{D}\llbracket n \rrbracket\,\Gamma\,\rho\,(\Box\ \oplus e_2 : \mathcal{R}) & = & \mathcal{D}\llbracket e_2 \rrbracket\,\Gamma\,\rho\,(n \oplus \Box : \mathcal{R}) & \text{(R10)}\\
\mathcal{D}\llbracket e_1 \oplus e_2 \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{D}\llbracket e_1 \rrbracket\,\Gamma\,\rho\,(\Box\ \oplus e_2 : \mathcal{R}) & \text{(R11)}\\
\mathcal{D}\llbracket e_1\,e_2 \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{D}\llbracket e_1 \rrbracket\,\Gamma\,\rho\,(\Box\ e_2 : \mathcal{R}) & \text{(R12)}\\
\mathcal{D}\llbracket \textbf{case}\,e\ \textbf{of}\,\{p_i \to e_i\} \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{D}\llbracket e \rrbracket\,\Gamma\,\rho\,(\textbf{case}\ \Box\ \ \textbf{of}\,\{p_i \to e_i\} : \mathcal{R}) & \text{(R13)}
\end{array}$$

Fallthrough

$$\begin{array}{lcll}
\mathcal{D}\llbracket e \rrbracket\,\Gamma\,\rho\,\mathcal{R} & = & \mathcal{B}\llbracket e \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R14)}
\end{array}$$

Rebuilding Expressions

$$\begin{array}{lcll}
\mathcal{B}\llbracket e' \rrbracket\,\Gamma\,\rho\,(\Box\ \oplus e_2 : \mathcal{R}) & = & \mathcal{B}\llbracket e' \oplus (\mathcal{D}\llbracket e_2 \rrbracket\,\emptyset\,\rho\,\epsilon) \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R15)}\\
\mathcal{B}\llbracket e' \rrbracket\,\Gamma\,\rho\,(n \oplus \Box : \mathcal{R}) & = & \mathcal{B}\llbracket n \oplus e' \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R16)}\\
\mathcal{B}\llbracket e' \rrbracket\,\Gamma\,\rho\,(\Box\, e : \mathcal{R}) & = & \mathcal{B}\llbracket e'(\mathcal{D}\llbracket e \rrbracket\,\emptyset\,\rho\,\epsilon) \rrbracket\,\Gamma\,\rho\,\mathcal{R} & \text{(R17)}\\
\mathcal{B}\llbracket x' \rrbracket\,\Gamma\,\rho\,(\textbf{case}\ \Box\ \ \textbf{of}\,\{p_i \to e_i\} : \mathcal{R}) & = & \textbf{let}\,\mathcal{D}\llbracket \Gamma|x' \rrbracket\,\emptyset\,\rho\,\epsilon\,\textbf{in} & \text{(R18)}\\
 & & \textbf{case}\,x'\ \ \textbf{of}\,\{p_i \to \mathcal{D}\llbracket \textbf{let}\,\Gamma\backslash x'\,\textbf{in}\,[p_i/x'][e_i]\mathcal{R} \rrbracket\,\emptyset\,\rho\,\epsilon\} & \\
\mathcal{B}\llbracket e' \rrbracket\,\Gamma\,\rho\,(\textbf{case}\ \Box\ \ \textbf{of}\,\{p_i \to e_i\} : \mathcal{R}) & = & \textbf{let}\,\mathcal{D}\llbracket \Gamma|\mathit{fv}(e') \rrbracket\,\emptyset\,\rho\,\epsilon\,\textbf{in} & \text{(R19)}\\
 & & \textbf{case}\,e'\ \textbf{of}\,\{p_i \to \mathcal{D}\llbracket \textbf{let}\,\Gamma\backslash\mathit{fv}(e')\,\textbf{in}\,[e_i]\mathcal{R} \rrbracket\,\emptyset\,\rho\,\epsilon\} & \\
\mathcal{B}\llbracket e' \rrbracket\,\Gamma\,\rho\,\epsilon & = & \textbf{let}\,\mathcal{D}\llbracket \Gamma \rrbracket\,\emptyset\,\rho\,\epsilon\,\textbf{in}\,e' & \text{(R20)}
\end{array}$$

Figure 4.3: Stronger driving and building algorithm

4.8.1.1 Positive Supercompilation

Our revised driving algorithm is defined in Figure 4.3 and our revised $\mathcal{D}_{app}(\,)$ is defined in Figure 4.4. An additional parameter appear as subscript to the rules: an ordered set Γ of expressions bound to variables ($x_1 = e_1,\ x_2 = e_2,\ \ldots$). We use the short-hand notation **let** Γ **in** e for **let** $x_1 = e_1$ **in let** $x_2 = e_2$ **in** .. **in** e. This revised algorithm changes the assumption about the shape of expressions when divide is called: the revised form is $\lambda\overline{x}.\textbf{let}\,\Gamma\,\textbf{in}\,[e]\mathcal{R}$.

Rule R18 and rule R19 uses $\mid$ for partitioning Γ: $\Gamma|x$ is the smallest prefix of Γ that is

$$\begin{array}{llll}
\mathcal{D}_{app}(g)\,\Gamma\,\rho\,\mathcal{R} & \mid \exists h.\rho|_g(h) \equiv \lambda\overline{x}.\mathbf{let}\,\Gamma\,\mathbf{in}\,[g]\mathcal{R} & = h\,\overline{x} & (1)\\
 & \mid \exists h.\rho|_g(h) \trianglelefteq \lambda\overline{x}.\mathbf{let}\,\Gamma\,\mathbf{in}\,[g]\mathcal{R} & & (2)\\
 & ,(f_g, [\overline{f}/\overline{y}]) = \mathit{divide}(\lambda\overline{x}.\mathbf{let}\,\Gamma\,\mathbf{in}\,[g]\mathcal{R}, \rho(h)) & = [\mathcal{D}[\![\overline{f}]\!]\,\emptyset\,\rho\,\epsilon/\overline{y}]\mathcal{D}[\![f_g]\!]\,\emptyset\,\rho\,\epsilon & \\
 & \mid \text{otherwise} & = \mathbf{letrec}\,h = \lambda.\overline{x}.\mathcal{D}[\![e]\!]\,\Gamma\,\rho'\,\mathcal{R} & (3)\\
 & & \quad \mathbf{in}\,h\,\overline{x} &
\end{array}$$

$$\begin{array}{ll}
\text{where } g \overset{\text{def}}{=} v & \\
 & \rho' = (h, \lambda\overline{x}.\mathbf{let}\,\Gamma\,\mathbf{in}\,[g]\mathcal{R}) : \rho\\
 & h \text{ fresh}\\
 & \overline{x} = \mathit{fv}(\mathbf{let}\,\Gamma\,\mathbf{in}\,[g]\mathcal{R})
\end{array}$$

Figure 4.4: Driving of applications

necessary to define *x*. We also introduce \ to perform the opposite operation: $\Gamma\backslash x$ is the largest suffix not necessary to define *x*. Rule R9 uses ;, which we define as:

$$\begin{array}{rcl}
(\Gamma, y = e); x = e' & \overset{\text{def}}{=} & (\Gamma; x = e'), y = e \text{ if y} \notin \mathit{fv}(e')\\
 & \overset{\text{def}}{=} & (\Gamma, y = e, x = e') \text{ otherwise}
\end{array}$$

The key idea in this improved supercompilation algorithm is to float let-expressions into the branches of case-expressions. We accomplish this by adding the bound expressions from let-expressions to our binding set Γ in rule R9. We make sure that we do not change the order between definition and usage of variables in rules R18 and R19 by extracting the necessary bindings outside of the case-expression, and the remaining independent bindings are brought into all the branches along with the surrounding context $\mathcal{R}$.

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

4.8.1.2 Removing Unnecessary List Traversals

The first example in Section 4.8.1 showed that there might be let-expressions in case-branches where the computed results are never used in the branch. This gives worse runtime performance than necessary since more intermediate results have to be computed, and also increases the compilation time since there are more expressions to transform. The only reason to have these let-expressions is to preserve the termination properties of the input program.

We could remove these superfluous let-expressions if we knew that they were terminating, something that would save both transformation time and execution time. It is clear that termination is undecidable in general, but our experience is that the functions that appear in practice are often recursive over the input structure. Functions with this property are quite well suited for termination analysis, for example the size-change principle (Lee et al. 2001; Sereni 2007).

Given a function *terminates*(e) that returns true if the expression e terminates, we can augment the let-rule (R9) to incorporate the termination information and discard such expressions,

$$\begin{array}{lcll}\mathcal{D}[\![\textbf{let}\, x = e_1 \,\textbf{in}\, e_2]\!]\, \Gamma\, \rho\, \mathcal{R} & = & \mathcal{D}[\![\textbf{let}\, \Gamma \,\textbf{in}\, [e_2]\mathcal{R}]\!]\, \emptyset\, \rho\, \epsilon & \text{if } \textit{terminates}(e_1) \text{ and } x \notin \textit{fv}(e_2) \\ & & \mathcal{D}[\![[e_1/x]e_2]\!]\, \Gamma\, \rho\, \mathcal{R} & \text{if } x \in \textit{strict}(e_2) \text{ and } x \in \textit{linear}(e_2) \\ & & \mathcal{D}[\![e_2]\!]\, (\Gamma; x = e_1)\, \rho\, \mathcal{R} & \text{otherwise}\end{array}$$

Figure 4.5: Extended Let-rule (R9)

shown in Figure 4.5. This allows the supercompiler to discard unused expressions, i.e dead code, which saves both transformation time and runtime.

Since we leave the choice of termination analysis open, it is hard to discuss scalability in general. The size-change principle has been used with good results in partial evaluation of large logic programs (Leuschel and Vidal 2009) and there are also polynomial time algorithms for approximating termination (Ben-Amram and Lee 2007).

4.8.1.3 Performance and Limitations

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

This supercompiler is stronger at the cost of larger expressions to test for the whistle. We estimate that our current work ends up somewhere between Supero (Mitchell 2008) and the algorithm defined in Section 2.3 with respect to transformation time since we are testing smaller expressions than Supero, at the expense of runtime performance.

For the second dimension: we have shown that the output from our supercompiler does not contain intermediate structures by manual transformations in Section 4.8.1. It is reasonable to conclude that these programs would perform better in a microbenchmark. We leave the question of performance of large real world programs open.

A limitation of our work is that there are still examples that our algorithm does not give the desired output for. Given **let** $x = (\lambda y.y)\ 1$ **in** $(x,\ x)$ a human can see that the result after transformation should be $(1,\ 1)$, but our supercompiler will produce **let** $x = 1$ **in** $(x,\ x)$. Mitchell (2008)[Sec 4.2.2] has a strategy to handle this, but we have not been able to incorporate his solution without severely increasing the amount of testing for non-termination done with the homeomorphic embedding. The reason we can not transform $(\lambda y.y)\ 1$ in isolation and then substitute the result is that the result might contain freshly generated function names, which might cause the supercompiler to loop.

4.8.2 Static Argument Transformation

The static argument transformation (Santos 1995) removes arguments that do not change in recursive calls. Instead of having these arguments as parameters to the function they are free variables that are accessed through the closure of the function. Santos performed measurements on an implementation inside GHC and showed that the transformation could give speedups. Some conditions on when to apply this transformation were given, and it was also noted that lambda lifting (Johnsson 1985) was in conflict with this transformation. More recently Bolingbroke (2009) has been investigating the feasibility of SAT and measured the effects of the transformation in a modern version of GHC.

We borrow an example from Santos (1995), the function *foldr* which is used throughout the Prelude in GHC to enable short cut deforestation (Gill 1996). Performing SAT on *foldr* gives a version with a local definition without the first and second argument:

$$
\begin{array}{l}
\mathit{foldr}\ f\ z\ l = \textbf{letrec}\ f'\ l' \ = \ \textbf{case}\ l'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [] \ \rightarrow \ z \\
\qquad\qquad\qquad\qquad\qquad\qquad (a : as) \ \rightarrow \ \textbf{let}\ v \ = \ f'\ as \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ f\ a\ v \\
\qquad\qquad\quad \textbf{in}\ f'\ l
\end{array}
$$

The transformed function has different operational properties than the original:

- Fewer arguments to be pushed to the stack in the recursive calls
- It exposes the possibility of inlining the function since it is not recursive any longer.
- Fewer free variables in the *v* closure: the original closure has three free variables (*f*, z and *as*) and the transformed closure has two (*f'* and *as*). In GHC this decreases the size of the closure, which affects performance.
- If there were subexpressions only referring to *f* and z the full laziness transformation could lift those expressions out of the recursive loop, avoiding recalculations each iteration.
- One more extra closure for the local recursive function.

This program transformation is sometimes beneficial, but characterizing those exact cases is left for future work. Our contribution is a transformation that is valid for both call-by-value and call-by-name. Implementing this transformation in the Timber compiler (Nordlander et al. 2008) is not going to give a speedup: the backend makes sure to lambda lift all functions before code generation.

Our example is mainly useful to demonstrate how our modified supercompiler works. As we will see, the majority of the transformation steps are already familiar the reader, but we will implicitly name expressions and assemble functions "on the way up" in the alogrithm. To transform the expression $\mathit{map}\ f'\ ys$ we start by allocating a new fresh function name (h_0) and inline the body of map:

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

There is nothing to be done for *ys*, so we continue to work on the branches of the case expression. Up until this point there is no difference at all in the transformation compared to the standard supercompiler, but the interesting part for the static argument transformation comes here: since we know the expression that we started out with we can compare the difference of the free variables in the starting expression (*f'* and *ys*) and the free variables in the map-expression in the second branch (*f'* and *xs*). Obviously *f'* is the same in both, so we can omit it the recursive call:

$$\begin{array}{llll}
\mathcal{D}_{app}(g)\,\rho\,\mathcal{R} & \mid \exists(h,\overline{y})\in\{(h,\overline{y})\mid \overline{y}\subseteq\overline{x}, \exists h.\rho(h)\equiv\lambda\overline{y}.[g]\mathcal{R}\} & = h\,\overline{y} & (1)\\
 & \mid \exists h.\rho|_g(h) \trianglelefteq \lambda\overline{x}.[g]\mathcal{R} & & (2)\\
 & ,(f_g,[\overline{f}/\overline{y}]) = divide(\lambda\overline{x}.[g]\mathcal{R},\rho(h)) & = [\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon/\overline{y}]\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon & \\
 & \mid \text{otherwise} & = \mathbf{letrec}\, \mathit{defs}\, \mathbf{in}\, h\,\overline{x} & (3)\\
\multicolumn{4}{l}{\quad \text{where } g \overset{\text{def}}{=} e}\\
\multicolumn{4}{l}{\qquad\quad e' = \mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R}}\\
\multicolumn{4}{l}{\qquad\quad \overline{x} = \mathit{fv}([g]\mathcal{R})}\\
\multicolumn{4}{l}{\qquad\quad \rho' = \{(h_{\overline{y}},\lambda\overline{y}.[g]\mathcal{R})\mid\overline{y}\subseteq\overline{x}\}\cup\rho}\\
\multicolumn{4}{l}{\qquad\quad \mathit{defs} = \{h_{\overline{y}} = \lambda\overline{y}.e'\mid\overline{y}\subseteq\overline{x}\}, h_i \text{ fresh}}
\end{array}$$

Figure 4.6: Modified driving of applications

$$\begin{array}{lll}
\mathbf{letrec}\ h_0\ ys & = & \mathbf{case}\ ys\ \mathbf{of}\\
 & & \quad [\,] \rightarrow [\,]\\
 & & \quad (x : xs) \rightarrow f'\ x : h_0\ xs\\
\mathbf{in}\ h_0\ ys & &
\end{array}$$

However, this example does not convey the complete picture: with a different example we might need several function definitions taking different subsets of the free variables. We therefore reserve more than one name when we name expressions, in fact we reserve one name for each possible subset of the free variables of the expression, which corresponds to the powerset of the free variables. We use the free variables to index into the set of h-functions. Figure 4.6 shows the updated $\mathcal{D}_{app}(\,)$ which also performs SAT, with changes to alternative 1 and 3. The expressions compared up to alpha renaming in alternative 1 might contain free variables, in which case there has to be an exact match of variable names for the match to occur. No other parts of the algorithm need to be modified.

Considering the similarity between the transformations it is not surprising that a slightly tweaked positive supercompiler can subsume SAT. The problem of characterizing when SAT is beneficial remains: what we show is that given a positive supercompiler it is possible and cheap to modify it to do SAT as well.

The transformation creates a group of mutually recursive functions for each call, so duplicated code will occur in the output program. Not all functions will have multiple recursive calls with different static arguments though, so it is possible to run a postprocessor on the output that removes some of these functions. Functions might have multiple recursive calls with different static arguments, something that effectively could remove locality properties and making the performance worse. Which function to call first is chosen non-deterministically, but it does not matter which function we start out with: recursive calls will call the right h_i with proper arguments.

The very purpose of this transformation is to create local recursive functions that contain free variables. Unfortunately this falls outside the scope of the improvement theory (Sands 1997) we use for proving the correctness of our supercompiler in Chapter 3, and another proof technical detail is that the memoization list ρ does not contain closed expressions any longer.

4.8.3 Preventing Exponential Code Growth

The following program, due to Simon Peyton Jones, will intuitively cause the supercompiler to inline "too much", despite never triggering the whistle:

$$\begin{aligned}
&\mathit{main}\ x\ =\ \mathit{print}\ (f_1\ x)\\
\\
&f_1\ x\ =\ (f_2\ (x\ +\ 1),\ f_2\ (x\ -\ 1))\\
&f_2\ x\ =\ (f_3\ (x\ +\ 1),\ f_3\ (x\ -\ 1))\\
&\ldots\\
&f_n\ x\ =\ e
\end{aligned}$$

For f_2 there will be two specialized copies in the output for the two different parameters: $x\ +\ 1$ and $x\ -\ 1$. For f_3 there will be four specialized copies for the parameters: $((x\ +\ 1)\ +\ 1)$, $x\ +\ 1$, $((x\ -\ 1)\ +\ 1)$, and $x\ -\ 1$. For f_4 there will be eight specialized copies, and for large *i*'s the number of specialized copies of f_i will be huge. An informal characterization of the above problem is that there are too many specialized functions and each one only gives a marginal performance benefit at best. The boring expressions that we described in Section 4.1 solves this particular instance of the problem, but we can augment each f_i with an extra parameter that is unused in the body which will avoid classifying the calls to f_i as boring:

$$\begin{aligned}
&\mathit{main}\ x\ =\ \mathit{print}\ (f_1\ x\ \mathit{Nothing})\\
\\
&f_1\ x\ y\ =\ (f_2\ (x\ +\ 1)\ \mathit{Nothing},\ f_2\ (x\ -\ 1)\ \mathit{Nothing})\\
&\ldots
\end{aligned}$$

This modified program will give rise to the same exponential code growth as the initial program even with the detection of boring expressions enabled in the supercompiler.

If our supercompiler could somehow detect that too many useless specializations were created during the transformation, and in that case avoid creating these specializations, this problem would disappear. An example that covers the decision points for this strategy could be:

$$\begin{aligned}
&\ldots\\
\\
&(\mathcal{D}[\![f_2\ (x\ +\ 1)]\!]\ \rho\,\epsilon,\ \mathcal{D}[\![f_2\ (x\ -\ 1)]\!]\ \rho\,\epsilon)
\end{aligned}$$

Supercompile the first component of the pair to get a new function h_1 and insert a call to that function:

$$(h_1\ x,\ \mathcal{D}[\![f_2\ (x\ -\ 1)]\!]\ \rho\,\epsilon)$$

Detect that the second component of the pair is similar to the first component. Propagate the expression $f_2\ (x-1)$ while backtracking the supercompiler to the state before supercompiling the first component:

$$\ldots$$

$$(\mathcal{D}[\![f_2\,(x\;+\;1)]\!]\,\rho\,\epsilon,\;\mathcal{D}[\![f_2\,(x\;-\;1)]\!]\,\rho\,\epsilon)$$

Generalize the first component of the pair against the expression that caused the backtracking, giving the generalized expression $f_2\,z$ and the substitution $[(x+1)/z]$ to supercompile:

$$([\mathcal{D}[\![x\;+\;1]\!]\,\rho\,\epsilon/z]\mathcal{D}[\![f_2\,z]\!]\,\rho\,\epsilon,\;\mathcal{D}[\![f_2\,(x\;-\;1)]\!]\,\rho\,\epsilon)$$

Supercompile the generalized first component of the pair to get a new function h_2 and insert a call to that function, as well as apply the supercompiled substitution:

$$(h_2\,(x\;+\;1),\;\mathcal{D}[\![f_2\,(x\;-\;1)]\!]\,\rho\,\epsilon)$$

This puts us back at almost the same state as previously, but the second component of the pair is an instance of the expression that we supercompiled to get h_2: $f_2\,z$. Immediately fold against that instance, instead of creating a new specialization:

$$(h_2\,(x\;+\;1),\;h_2\,(x\;-\;1))$$

It turns out that we are already familiar with the tools needed to detect the above scenario and perform the backtracking necessary to prevent the code growth. We introduce the classical most specific generalization (*msg*) (Lassez et al. 1988) that we mentioned in Chapter 2:

Definition 4.1 (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* e' *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 $\theta_2 t_g \equiv e'$.*
- *A* most specific generalization *(msg) of two terms* e *and* e' *is a generalization $(t_g, \theta_1, \theta_2)$ such that for every other generalization $(t'_g, \theta'_1, \theta'_2)$ of* e *and* e' *it holds that t_g is an instance of t'_g.*

We also introduce a grammar for something we label simple expressions:

$$s ::= x \mid n \mid s \oplus s$$

This allows us to give a definition of when two expressions are similar, which will be used to detect when the code growth scenario just described is happening. Once the code growth is detected, the supercompiler will generalize the expressions and use the same specialization for several different instances that are similar to each other.

Definition 4.2 (Similar expressions)**.** *Two expressions $[g]\mathcal{R}$ and $[g]\mathcal{R}'$ are similar, $[g]\mathcal{R} \approx [g]\mathcal{R}'$, if:*

- *length $\mathcal{R}$ = length $\mathcal{R}'$*
- *the respective frames of $\mathcal{R}$ and $\mathcal{R}'$ are the same type (lock-step shallow equality)*
- *given $(t_g, [e_1/x_1, \ldots, e_n/x_n], [e'_1/x_1, \ldots, e'_n/x_n]) = msg\,[g]\mathcal{R}\,[g]\mathcal{R}'$ then all e_i and e'_i are simple expressions*

The first two tests, comparing the length of the contexts and comparing the type of each frame, can be performed at the same time with just one traversal of the contexts. An even cheaper test would be to cache the length of each context $\mathcal{R}$ in ρ and use this information to prune the possible candidates in order to avoid comparing contexts that are of different lengths.

There are at least two problems here: the backtracking mechanism does not come for free and complicates the algorithm, and as shown in the example it is sometimes necessary to backtrack "sideways" in the process tree. There is also a lot of bookkeeping to do, but the plumbing necessary for the bookkeeping is essentially there already as we have a state monad at our disposal.

This extension reduces code size, and can reduce the amount of work performed during the transformation since it decreases the number of specializations. However, it can also increase compilation times since the backtracking mechanism discards transformation work that is sometimes independent of the generalized expressions, thereby forcing the supercompiler to perform the same transformation again. Whether it is better to perform some backtracking at compilation time to avoid performing exponential work in the supercompiler is dependent on the particular scenario, and we leave it for future work to characterize when this is beneficial or not. The definition of simple and similar expressions could also be altered, which would change how often generalizations are performed and how much backtracking that is necessary.

4.8.4 Impure Languages

Inspired by Sestoft's (2010b) work on performing partial evaluation of spreadsheet-defined functions we naturally asked ourselves whether it was possible to define a supercompilation algorithm for spreadsheet-defined functions as well. People in general do not think of spreadsheets as a programming language, but Peyton Jones et al. (2003) convincingly argue that it is indeed a very limited functional language, and Sestoft (2010a) points to the existance of biological models defined in spreadsheets that takes hours to load. Beyond the popularity of spreadsheets, there is also a technical motivation for solving this problem: the programming language used in spreadsheet software is very restricted, but it *does* contain some effectful primitive operations (such as *RAND()*), so solving this problem would bring us one step closer to the solution for supercompiling other effectful languages, such as OCaml or F#.

The crucial property of these primitive effectful operations is that they are commutative, which makes it safe to reorder computations. An example of this property are the two expressions **let** $x = \mathit{RAND}()$ **in** $\mathit{RAND}() + x$ and $\mathit{RAND}() + \mathit{RAND}()$, which are observationally equivalent.

To capture this we augment our previous call-by-value language with a primitive commutative effectful operations δ, as defined in Figure 4.7. We use $\Theta \vdash \delta\,\overline{v} \downarrow \Theta', v$ to model the effect and return a new value *v* in our operational semantics which we define in Figure 4.8.

Definition 4.3 (Commutative effects). *Two effects $\delta_1\,\overline{v}_1$ and $\delta_2\,\overline{v}_2$ are commutative iff whenever $\Theta \vdash \delta_1\,\overline{v}_1 \downarrow \Theta_1, v_1$ and $\Theta_1 \vdash \delta_2\,\overline{v}_2 \downarrow \Theta', v_2$ there exists a Θ_2 such that $\Theta \vdash \delta_2\,\overline{v}_2 \downarrow \Theta_2, v_2$ and $\Theta_2 \vdash \delta_1\,\overline{v}_1 \downarrow \Theta', v_1$.*

It turns out that the algorithm defined in Section 4.8.1 is sufficient to supercompile this language, and preserve the semantics of its input. We are however unable to prove this since

Expressions

$$\begin{array}{lcl} e & ::= & n \mid x \mid g \mid e\,e' \mid \lambda x.e \mid k\,\overline{e} \mid \delta\,\overline{e} \mid e_1 \oplus e_2 \mid \textbf{case}\,e\ \textbf{of}\ \{p_i \to e_i\} \\ & \mid & \textbf{let}\,x = e\ \textbf{in}\,e' \end{array}$$

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

Values

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

Figure 4.7: The language

Call-by-value reduction contexts

$$\begin{array}{lcl} \mathcal{E} & ::= & \Box \mid \mathcal{E}\,e \mid (\lambda x.e)\,\mathcal{E} \mid k\,\overline{\mathcal{E}} \mid \delta\,\overline{\mathcal{E}} \mid \mathcal{E} \oplus e \mid n \oplus \mathcal{E} \mid \textbf{case}\ \mathcal{E}\ \textbf{of}\ \{p_i \to e_i\} \\ & \mid & \textbf{let}\,x = \mathcal{E}\ \textbf{in}\,e \end{array}$$

Call-by-value evaluation relation

$$\begin{array}{lcll} \Theta \vdash [g]\mathcal{E} & \mapsto & \Theta \vdash [v]\mathcal{E}, \text{ if } g \overset{\text{def}}{=} v & \text{(Global)} \\ \Theta \vdash [\delta\,\overline{v}]\mathcal{E} & \mapsto & \Theta' \vdash [v]\mathcal{E}, \text{ if } \Theta \vdash \delta\,\overline{v} \downarrow \Theta', v & \text{(Effect)} \\ \Theta \vdash [(\lambda x.e)\,v]\mathcal{E} & \mapsto & \Theta \vdash [[v/x]e]\mathcal{E} & \text{(Beta)} \\ \Theta \vdash [\textbf{let}\,x = v\,\textbf{in}\,e]\mathcal{E} & \mapsto & \Theta \vdash [[v/x]e]\mathcal{E} & \text{(Let)} \\ \Theta \vdash [\textbf{case}\,k\,\overline{v}\,\textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}]\mathcal{E} & \mapsto & \Theta \vdash [[\overline{v}/\overline{x}_j]e_j]\mathcal{E}, \text{ if } k = k_j & \text{(KCase)} \\ \Theta \vdash [\textbf{case}\,n\,\textbf{of}\ \{n_i \to e_i\}]\mathcal{E} & \mapsto & \Theta \vdash [e_j]\mathcal{E}, \text{ if } n = n_j & \text{(NCase)} \\ \Theta \vdash [n_1 \oplus n_2]\mathcal{E} & \mapsto & \Theta \vdash [n]\mathcal{E}, \text{ if } n = n_1 + n_2 & \text{(Arith)} \end{array}$$

Figure 4.8: Effectful call-by-value reduction semantics

this effectful language falls outside the scope of the improvement theory (Sands 1997), but intend to investigate this further in our future work.

The algorithm will reorder computations and this is perfectly safe since all effects are commutative, but computations will never be duplicated since that might change the meaning of the program. The previous motivation for not duplicating computations was efficency of the output program, but it is now essential for correctness in order to preserve the semantics of the program.

CHAPTER 5

Implementation and Measurements

During the course of our implementation we re-discovered known tricks that makes the implementation more convenient, and also some new tricks that we have not found in the literature. We start this chapter by describing these tricks hoping they will be useful for any future researcher on supercompilation. We also give measurements from two implementations of the algorithm:

- The basic call-by-value algorithm described in Chapter 2 inside the Timber compiler on a set of examples from the literature of deforestation in Section 5.5
- The algorithm described in Chapter 4 inside GHC using the standard NoFib benchmark suite (Partain 1992) in Section 5.6

Our design allows us to supercompile most programs from the spectral part of nofib in less than three seconds. Some of these programs are 10 times larger than previously supercompiled Haskell programs. We do not believe that an ordinary programmer will turn on the supercompiler for his development builds, but it is certainly reasonable to do so for testing and nightly builds.

5.1 Parallelism

The supercompiler is currently defined in a very sequential way: it is not possible to inspect the store unless the previous parts of the expression is supercompiled and the store contains all the new definitions. Beyond a lot of infrastructure hacking it appears that lifting this restriction while preserving code sharing is an interesting algorithmic problem that probably requires some research.

There are however some parts of the supercompiler as it is currently defined that are inherently parallel: the testing for folding and non-termination in $\mathcal{D}_{app}()$ can easily be done in parallel since the common case is that there are several possible candidates to test against. This gave a measurable speedup in an an earlier version of our prototype but had to be removed because other parts of GHC are not threadsafe.

5.2 Name Capture

Avoiding name capture can cause significant churn and make the supercompiler dreadfully slow. Bolingbroke and Peyton Jones (2010) report that for a particular example 42% of their supercompilation time is spent on managing names and renaming.

Our current implementation uses eager substitutions with the scheme outlined by Peyton Jones and Marlow (2002). There are two obvious improvements to this: using a lazy substitution that the supercompiler applies as it traverses the expression would avoid some churn, and this could be further reduced by allowing some shadowing during supercompilation.

5.3 Speed Improvements of the Homeomorphic Embedding

Our implementation stores expressions annotated with their size in the memoization list. These annotations can be used for avoiding some testing altogether: an expression can only fold against an expression of equal size and an expression can never be homeomorphically embedded in a smaller expression. A similar representation was also independently suggested by Ilya Klyuchnikov so this seems to be a well known optimization in the supercompilation community.

5.4 Splitting Typed Terms

The most obvious difference between our implementation and the algorithm described in this paper is that the intermediate language of GHC is a typed language (Sulzmann et al. 2007). This brings additional complexity when generalizing similar terms of different type against each other.

Pfenning (1991) has already identified our problems: the presence of binders in expressions and the presence of types in binders. There are two advantages for our particular setting: 1) type variables are the top element for types; and 2) it is safe to fail during generalization and call split on the term instead.

After having implemented the type generalization we noticed that it was not performed very often in practice on our examples from nofib. The main cost related to this feature appears to be the implementation cost.

5.5 Timber Measurements

In this section we provide measurements on a set of common examples from the literature on deforestation and perform a detailed analysis for each example. We show that our positive supercompiler from Chapter 2.2, with a small change to the folding mechanism as detailed in Figure 5.1 and augmented with the store from Chapter 4, removes intermediate structures and can improve the performance by an order of magnitude for certain benchmarks. The supercompiler was implemented as a pass in the Timber compiler (Nordlander et al. 2008). Timber

$$
\begin{array}{lllr}
\mathcal{D}_{app}(g)\,\rho\,\mathcal{R} & \mid \exists h.\rho|_g(h) \equiv \lambda\overline{x}.[g]\mathcal{R} & = h\,\overline{x} & (1)\\
 & \mid \exists h.\rho|_g(h) \trianglelefteq \lambda\overline{x}.[g]\mathcal{R} & & (2)\\
 & ,(f_g, [\overline{f}/\overline{y}]) = divide(\lambda\overline{x}.[g]\mathcal{R}, \rho(h)) & = [\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon/\overline{y}]\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon & \\
 & \mid h \in e' & = \textbf{letrec}\,h = \lambda\overline{x}.e'\,\textbf{in}\,h\,\overline{x} & (3a)\\
 & \mid \text{otherwise} & = e' & (3b)\\
\multicolumn{4}{l}{\quad\text{where } g \overset{\text{def}}{=} e}\\
 & \rho' = (h, \lambda\overline{x}.[g]\mathcal{R}) : \rho & & \\
 & h \text{ fresh} & & \\
 & \overline{x} = \mathit{fv}([g]\mathcal{R}) & & \\
 & e' = \mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R} & &
\end{array}
$$

Figure 5.1: Modified driving of applications

is a pure functional call-by-value language which is very close to the language we describe in Chapter 2. We have left out the full details of the instrumentation of the run-time system but it is available in a separate report (Jonsson 2008).

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

Raw data for the time and size measurements before and after supercompilation are shown in Table 5.1, and allocation measures in Table 5.2. Compilation times are shown in Table 5.3. The time column is the number of clock ticks obtained from the *RDTSC* instruction available on Intel/AMD processors, and the binary size is in bytes. The total number of allocations and the total memory size in bytes allocated by the program are displayed in their respective column. The compilation times are measured in seconds and times from left to right are for producing an object file, producing an executable binary, and the corresponding operations with supercompilation turned on.

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

The supercompiled results on these particular benchmarks are identical to the results reported in previous work for call-by-name languages by Wadler (1990) and Sørensen et al. (1996). We do not provide any execution-time comparisons with these, though, since for identical intermediate representations after supercompilation, such measurements would only illustrate differences caused by back-end implementation techniques.

5.5.1 Double Append

As previously seen, supercompiling the appending of three lists saves one traversal over the first list. This is an example by Wadler (1990), and the intermediate structure is fused away by our supercompiler. The program is:

	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 Squares of a Tree	276,102,012	28,737,648	95,452	104,912
Kort's Raytracer	12,050,880	7,969,224	91,968	91,460

Table 5.1: Time and size measurements for Timber

	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 Squares of a Tree	4,194,338	91	29,360,496	908
Kort's Raytracer	60,021	17	320,144	124

Table 5.2: Allocation measurements for Timber

$$
\begin{aligned}
&\mathit{append}\ \mathit{xs}\ \mathit{ys} &&= \textbf{case}\ \mathit{xs}\ \textbf{of} \\
& && \quad [] \ \rightarrow\ \mathit{ys} \\
& && \quad (x' : \mathit{xs}') \ \rightarrow\ x' : (\mathit{append}\ \mathit{xs}'\ \mathit{ys}) \\
&\mathit{main}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} &&= \mathit{append}\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})\ \mathit{zs}
\end{aligned}
$$

Supercompiling this program gives the same result that we obtained manually in Section 1.3:

$$
\begin{aligned}
&h_1\ \mathit{xs}_1\ \mathit{ys}_1\ \mathit{zs}_1 &&= \textbf{case}\ \mathit{xs}_1\ \textbf{of} \\
& && \quad [] \ \rightarrow\ \textbf{case}\ \mathit{ys}_1\ \textbf{of} \\
& && \qquad\qquad [] \ \rightarrow\ \mathit{zs}_1 \\
& && \qquad\qquad (y_1' : \mathit{ys}_1') \ \rightarrow\ y_1' : (h_2\ \mathit{ys}_1'\ \mathit{zs}_1) \\
& && \quad (x_1' : \mathit{xs}_1') \ \rightarrow\ x_1' : (h_1\ \mathit{xs}_1'\ \mathit{ys}_1\ \mathit{zs}_1) \\
&h_2\ \mathit{xs}_2\ \mathit{ys}_2 &&= \textbf{case}\ \mathit{xs}_2\ \textbf{of} \\
& && \quad [] \ \rightarrow\ \mathit{ys}_2 \\
& && \quad (x_2' : \mathit{xs}_2') \ \rightarrow\ x_2' : (h_2\ \mathit{xs}_2'\ \mathit{ys}_2)
\end{aligned}
$$

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

Table 5.3: Compilation times in seconds for Timber

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

In this measurement, three strings of 9000 characters each were appended to each other into a 27 000 character string. As can be seen in Table 5.2, the number of allocations goes down as one iteration over the first string is avoided. The binary size increases 1316 bytes, on a binary of roughly 90k.

5.5.2 Factorial

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

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

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

The program is transformed to:

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

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

One recursion and a couple of reductions are eliminated, thereby slightly reducing the runtime. The allocations remain the same and the final binary size remains unchanged.

5.5.3 Flip a Tree

Flipping a tree is another example by Wadler (1990), and just like Wadler we perform a double flip (thus restoring the original tree) before printing the total sum of all leaves.

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

$$\begin{aligned} &sumtr\ (Leaf\ a)\ =\ a \\ &sumtr\ (Branch\ l\ r)\ =\ sumtr\ l\ +\ sumtr\ r \end{aligned}$$

$$\begin{aligned} &flip\ (Leaf\ x)\ =\ Leaf\ x \\ &flip\ (Branch\ l\ r)\ =\ Branch\ (flip\ r)\ (flip\ l) \end{aligned}$$

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

This is transformed into:

$$\begin{array}{l}
h\ t\ =\ \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad \mathit{Leaf}\ d\ \rightarrow\ d \\
\qquad\qquad \mathit{Branch}\ l\ r\ \rightarrow\ (h\ l)\ +\ (h\ r) \\
\mathit{main}\ \mathit{xs}\ =\ \mathit{show}\ (\ \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Leaf}\ d\ \rightarrow\ d \\
\qquad\qquad\qquad\qquad \mathit{Branch}\ l\ r\ \rightarrow\ (h\ l)\ +\ (h\ r))
\end{array}$$

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

5.5.4 Sum of Squares of a Tree

Computing the sum of the squares of the data members of a tree is the final example by Wadler (1990).

$$\begin{array}{l}
\mathit{data}\ \mathit{Tree}\ a\ =\ \mathit{Leaf}\ a\ |\ \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
\\
\mathit{square}\ ::\ \mathit{Int}\ \rightarrow\ \mathit{Int} \\
\mathit{square}\ x\ =\ x * x \\
\\
\mathit{sumtr}\ (\mathit{Leaf}\ x)\ =\ x \\
\mathit{sumtr}\ (\mathit{Branch}\ l\ r)\ =\ \mathit{sumtr}\ l\ +\ \mathit{sumtr}\ r \\
\\
\mathit{squaretr}\ (\mathit{Leaf}\ x)\ =\ \mathit{Leaf}\ (\mathit{square}\ x) \\
\mathit{squaretr}\ (\mathit{Branch}\ l\ r)\ =\ \mathit{Branch}\ (\mathit{squaretr}\ l)\ (\mathit{squaretr}\ r) \\
\\
\mathit{main}\ \mathit{xs}\ =\ \mathit{show}\ (\mathit{sumtr}\ (\mathit{squaretr}\ \mathit{xs}))
\end{array}$$

This is transformed to:

$$\begin{array}{l}
h\ t\ =\ \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad \mathit{Leaf}\ d\ \rightarrow\ d\ *\ d \\
\qquad\qquad \mathit{Branch}\ l\ r\ \rightarrow\ (h\ l)\ +\ (h\ r) \\
\mathit{main}\ \mathit{xs}\ =\ \mathit{show}\ (\ \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Leaf}\ d\ \rightarrow\ d\ *\ d \\
\qquad\qquad\qquad\qquad \mathit{Branch}\ l\ r\ \rightarrow\ (h\ l)\ +\ (h\ r))
\end{array}$$

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

5.5.5 Kort's Raytracer

The inner loop of a raytracer (Kort 1996) written in Haskell is extracted and transformed.

$$\begin{aligned}
&\mathit{zipWith}\ f\ (x : xs)\ (y : ys) = (f\ x\ y) : \mathit{zipWith}\ f\ xs\ ys\\
&\mathit{zipWith}\ \_\ \_\ \_ = []\\
\\
&\mathit{sum} :: [\mathit{Int}] \rightarrow \mathit{Int}\\
&\mathit{sum}\ [] = 0\\
&\mathit{sum}\ (x : xs) = x + \mathit{sum}\ xs\\
\\
&\mathit{main}\ xs\ ys = \mathit{sum}\ (\mathit{zipWith}\ (*)\ xs\ ys)
\end{aligned}$$

The transformed result is:

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

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

5.6 Haskell Measurements

We start by establishing a base line against ghc -O2 on a small set of examples from the imaginary part of nofib as shown in Table 5.4. The table uses output from *nofib-analyse*, where each column indicates the difference between the baseline (O2 in this case) and the optimization. Absolute numbers without a sign, such as the runtime for rfib, means that the absolute numbers were too small to give reliable information. The size column is the size of the binary produced by ghc, which includes the runtime system and libraries. By turning off the features normalization of expressions, boring contexts, discarding expressions and tainting expressions in our supercompiler we get to something that is close to an ordinary positive supercompiler (Sørensen et al. 1996). The compilation times range between 5 and 12 seconds.

We suffer from the same problem that Bolingbroke and Peyton Jones (2010) report: the supercompiler can sometimes prevent GHC from applying other important optimizations such as unboxing of arithmetics. Since that is a slightly different algorithm it is not exactly the same programs that get staggering increases in memory allocations. A particular example is *tak*, which our supercompiler slightly optimizes, whereas their get an 18,000-fold increase in memory allocations.

However, our supercompiled *integrate* has a 65,000-fold increase in memory allocations, and the problem is that during supercompilation there is an intuitively "bad" expression split resulting in a higher order function being called repeatedly from inside the main loop. This is also what happens for *wheel-sieve2*, but the results are less severe.

There are two examples that show a rather hefty increase in binary size: *digits-of-e1* and *integrate*. One might argue that a 18% reduction in runtime is worth a size increase of 22.2%

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (s)
bernouilli	+15.0	-1.5	+0.0	+5.9	+0.0	6.60
digits-of-e1	+22.0	-20.3	-17.6	-16.8	+0.0	11.20
exp3_8	+12.7	-16.7	-16.7	-17.5	+0.0	5.53
integrate	+26.2	+65252.6	+105413.5	+93460.4	+32.4	11.95
primes	+12.7	+0.2	-16.7	-16.0	+0.0	5.30
rfib	+12.7	-1.0	0.11	0.11	+0.0	5.42
tak	+12.7	-12.5	+3.6	+3.4	+0.0	5.27
wheel-sieve2	+13.8	+307.0	0.55	+196.2	+252.1	6.27

Table 5.4: GHC -O2 compared with Supercompilation

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-7.6	+0.0	+0.0	-8.4	+0.0	-61.2
digits-of-e1	-6.9	+0.0	+1.9	+3.8	+0.0	-35.6
exp3_8	-7.5	+0.0	-0.6	+0.8	+0.0	-68.4
integrate	-6.8	+0.1	+0.2	+0.2	-4.4	-33.1
primes	-7.5	+0.0	+3.3	+2.5	+0.0	-69.2
rfib	-7.5	+0.0	0.11	0.11	+0.0	-70.1
tak	-7.5	+0.0	-3.8	-3.5	+0.0	-70.0
wheel-sieve2	-7.4	+0.0	-1.1	-1.3	+0.0	-63.2

Table 5.5: Supercompilation compared with Supercompilation+normalization in GHC

for *digits-of-e1*. This is obviously not the case with *integrate*, a massive slowdown combined with a 26% size increase of the resulting binary.

By turning on normalization of expressions (Table 5.5) there is a slight decrease of binary size, a fairly large decrease of compilation time and no changes to the performance of the output programs. This is a worthwhile improvement over regular positive supercompilation.

Table 5.6 shows that by turning on boring contexts it is possible to further shrink the binary size and reduce compilation time. Two benchmarks are affected by this performance wise: *digits-of-e1* is optimized with another 20% reduction in memory allocations giving a slight performance improvement while *tak* goes back to the same amount of memory allocations as without supercompilation and a slight performance increase.

By discarding expressions (Table 5.7) there is a further decrease of binary size and the performance loss *wheel-sieve2* suffered from plain supercompilation is recovered. The compilation times are all shorter except for *digits-of-e1* which supercompiles several expressions and discards them only to find them in a slightly different context at a later time and repeating this work.

Tainting expressions (Table 5.8) gives an additional decrease of compilation time for all but one benchmark, and two benchmarks show speedups: the performance of *integrate* is improved compared to ordinary optimization and *digits-of-e1* gets 2.5% faster as well. The compilation time for *exp3_8* more than doubles, but looking at the absolute numbers the compilation time

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-2.0	+0.0	+0.0	+0.8	+0.0	-20.1
digits-of-e1	-1.6	-22.6	-6.5	-8.1	+0.0	-15.2
exp3_8	-1.6	+0.0	-0.6	-0.5	+0.0	-41.1
integrate	-3.6	-0.1	-0.6	-0.6	+1.6	-22.3
primes	-1.6	+0.0	+0.0	+0.0	+0.0	-39.3
rfib	-1.6	+0.8	0.13	0.13	+0.0	-42.0
tak	-1.6	+13.9	-2.1	-1.8	+0.0	-42.7
wheel-sieve2	-1.6	+0.0	-0.4	-0.3	+0.0	-35.0

Table 5.6: Supercompilation + normalization compared with Supercompilation + normalization + boring expressions in GHC

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-1.6	-0.0	+0.0	+3.0	+0.0	-34.0
digits-of-e1	-10.0	+0.0	-5.0	-3.9	+0.0	+38.9
exp3_8	-2.5	+20.0	+21.4	+22.0	+0.0	-59.2
integrate	-11.6	-0.0	-0.0	-0.0	+0.8	-25.2
primes	-2.5	-0.2	+0.0	-0.6	+0.0	-59.6
rfib	-2.5	+0.1	0.12	0.12	+0.0	-61.7
tak	-2.5	+0.3	+5.1	+4.8	+0.0	-60.0
wheel-sieve2	-3.6	-75.4	-63.0	-62.4	-67.5	-56.0

Table 5.7: Supercompilation + normalization + boring expressions compared with Supercompilation + normalization + boring expressions + discarding expressions in GHC

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (%)
bernouilli	-2.2	+1.5	+0.0	-2.9	+0.0	-59.0
digits-of-e1	+0.0	+0.0	-2.6	-1.5	+0.0	-87.5
exp3_8	+0.0	+0.0	+0.0	+0.0	+0.0	+116.7
integrate	-0.3	-99.9	-99.9	-99.9	-22.7	-85.7
primes	+0.0	+0.0	+0.0	+0.0	+0.0	-5.0
rfib	+0.0	+0.0	0.13	0.13	+0.0	+0.0
tak	+0.0	+0.0	+0.0	-0.3	+0.0	+0.0
wheel-sieve2	+0.0	-0.1	0.19	-8.7	-12.7	-19.7

Table 5.8: Supercompilation + normalization + boring expressions + discarding expressions compared with Supercompilation + normalization + boring expressions + discarding expressions + tainting functions in GHC

Program	Size (%)	Allocs (%)	Runtime (%)	Elapsed (%)	TotalMem (%)	Compilation Time (s)
bernouilli	+0.3	-0.0	+0.0	-1.5	+0.0	0.55
digits-of-e1	+0.6	-38.3	-27.5	-25.6	+0.0	1.07
exp3_8	+0.0	-0.0	+0.0	+0.0	+0.0	0.91
integrate	+0.0	-52.6	-29.4	-27.6	+0.0	0.66
primes	+0.0	+0.0	-13.9	-12.3	+0.0	0.38
rfib	+0.0	+0.0	0.12	0.12	+0.0	0.36
tak	+0.0	+0.0	+2.6	+2.3	+0.0	0.36
wheel-sieve2	+0.0	+0.0	0.19	+0.0	+0.0	0.49
atom	+0.1	-4.4	-10.3	-10.1	+0.0	0.62
awards	+0.1	-1.2	0.00	0.00	+0.0	0.53
banner	+0.0	-1.6	0.00	0.00	+0.0	0.92
calendar	+2.1	-3.1	+0.0	-2.9	+0.0	1.32
circsim	+3.4	-1.5	+1.8	+0.0	-7.1	2.25
clausify	+0.3	-3.8	-6.9	-6.6	+0.0	0.56
constraints	+0.3	+1.3	-4.7	-4.4	-93.8	0.75
cryptarithm1	+0.0	+1.1	-2.6	+1.5	+0.0	0.36
fish	+3.8	-26.3	0.01	0.01	+0.0	1.55
gcd	+0.3	-2.5	0.09	0.09	+0.0	0.50
integer	+0.4	-6.5	-4.0	-3.8	-50.0	0.27
life	+1.0	-37.8	0.08	0.08	+0.0	0.73
pretty	+0.1	-3.4	0.00	0.00	+0.0	0.61
primetest	+0.0	-0.0	+0.0	-1.4	+0.0	0.64
scc	+0.0	+0.0	0.00	0.00	+0.0	0.33
sorting	+0.0	+0.0	0.00	0.00	+0.0	0.58

Table 5.9: GHC -O2 compared with Supercompilation with all features enabled

remains under one second.

We finally give the full table with a greater selection of benchmarks from both the spectral and imaginary parts of nofib in Table 5.9. *Fish* is the benchmark with the biggest size increase at 3.8%, but the supercompiler also manages to remove one fourth of the total allocations.

Circsim is another interesting example, because it consists of about 400 lines of code and compiles in 2.25 seconds with the supercompiler turned on. The supercompiler manages to reduce the memory allocations slightly and give 3.4% binary size increase.

We saw that before adding the tainting of functions it was possible for the supercompiler to perform a lot of unnecessary work by speculatively supercompiling an expression, discard the result, supercompile the subexpressions, discard those results, and so on. One possible strategy to mitigate this further is to keep the supercompiled result around as a cache that successive supercompilation of subexpressions can fold against, thereby avoiding to repeat work.

Our experience so far is that the discarding of functions is not preventing optimizations, on the contrary: programs that were optimized by the supercompiler without discarding expressions are actually slightly faster when discarding the large useless specializations that ended up in the program previously.

CHAPTER 6

Related Work

This chapter will give an overview of the state of the art in program specialization. The comparison and critical analysis appears in Section 6.9.

6.1 Deforestation

Deforestation is a slightly weaker transformation than supercompilation (Sørensen et al. 1994) but deforestation algorithms for call-by-name languages can still remove all intermediate structures from the examples given in Sections 1.3, 2.3, and 4.8.1.

Deforestation was pioneered by Wadler (1990) for a first order language more than fifteen years ago. The initial deforestation had support for higher order macros which are incapable of fully emulating higher order functions.

Marlow and Wadler (1992) addressed the first-order restriction and presented a deforestation algorithm for a higher order language. This work was refined in Marlow's (1995) dissertation, where he also related deforestation to the cut-elimination principle of logic. Chin (1994) has also generalised Wadler's deforestation to higher-order functional programs by using syntactic properties to decide which terms that can be fused.

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

Alimarine and Smetsers (2005) have improved the producer and consumer analyses in Chin's (1994) algorithm by basing them on semantics rather than syntax. They show that their algorithm can remove much of the overhead introduced from generic programming (Hinze 2000).

6.2 Supercompilation

Turchin (1986b) invented supercompilation in the context of the functional language Refal.

The supercompiler Scp4 (Nemytykh 2003) is implemented in Refal and is the most well-known implementation from this line of work. Sørensen et al. (1996) and Secher (1999); Secher and Sørensen (2000) presented supercompilers formulated for pure first order functional languages but never reported any measurements of the performance of the algorithms or the performance of the resulting programs.

Most work on supercompilation has been for languages with call-by-name semantics, with the notable exception of recent work by Bolingbroke and Peyton Jones (2010) for call-by-need languages. All call-by-need and call-by-name supercompilers succeed on the examples we outlined in Sections 1.3, 2.3, and 4.8.1, and are very close algorithmically to our current work.

Supercompilation (Turchin 1979, 1980, 1986a,b) is capable of both removing intermediate structures and perform partial evaluation as well as some other optimisations. Scp4 (Nemytykh 2003) is the most well-known implementation from this line of work.

The *positive supercompiler* (Sørensen et al. 1996) is a variant which only propagates positive information, such as equalities. The propagation is done by unification and the work highlights how similar deforestation and positive supercompilation really are. Narrowing-driven partial evaluation (Alpuente et al. 1998; Albert and Vidal 2001) is the functional logic programming equivalent of positive supercompilation but formulated as a term rewriting system. Their approach also deals with non-determinism from backtracking, which makes the corresponding algorithms more complicated.

Strengthening the information propagation mechanism to propagate not only positive but also negative information yields *perfect supercompilation* (Glück and Klimov 1993; Secher 1999; Secher and Sørensen 2000). Negative information is the opposite of positive information, namely inequalities. These inequalities can be used to prune case-expression branches known not to be applicable, for example.

More recently there has been a spur of activity in actual implementations of supercompilers:

Supero Supero (Mitchell and Runciman 2008; Mitchell 2008) can optimize a text book example of word counting written in Haskell to the same performance as a C program performing the same task. The latest version has changed the design significantly in order to improve performance of the supercompiler and Mitchell (2010) report compilation times below 4 seconds for all examples.

CHSC Bolingbroke and Peyton Jones (2010) presented a supercompiler with call-by-need semantics that did not require its input to be lambda-lifted. The details of the mechanism in their supercompiler that allows to avoid lambda-lifting is still unclear to us, and we need to investigate it further.

HOSC Klyuchnikov and Romanenko (2009, 2010) have implemented HOSC in Scala and used it for proving equivalence of terms. In more recent work Klyuchnikov (2010) shows that it is also possible to use HOSC for program optimization. The main difference from our work is the focus: we want performance above all else and have therefore weakened the positive supercompilation algorithm as much as we could, whereas HOSC goes in the opposite direction with a strengthened algorithm.

Others Reich et al. (2010) use supercompilation to optimize programs intended to run on the Reduceron, an FPGA-based soft processor for executing lazy functional programs. They report average performance boosts on a set of programs of 40%.

Supercompilation has also been used to verify the correctness of cache coherence protocols (Lisitsa and Nemytykh 2007). We do not believe that our call-by-value supercompiler is useful for proving term equivalence or correctness of cache coherence protocols since it is inherently weaker than the corresponding supercompiler with call-by-name semantics. Klyuchnikov and Romanenko (2011) notes that it is easier in some sense to transform two different programs into the same bad program than it is to transform them into the same good program. Under call-by-value semantics this might fail because of imprecision of the strictness analysis, preventing the two programs to reach the same syntactic form.

6.3 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.

6.4 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.

More recently partial evaluation has been performed in a dependently typed language to remove the interpretative overhead for embedded domain specific languages (EDSL's) (Brady and Hammond 2010). It would be an interesting exercise to try to get the same results with our supercompiler. We could drop the strictness requirement in rule R9 of our supercompiler in their setting since functions are known to always terminate, which would intuitively strengthen it.

Partial evaluators such as Ecce and Logen (Leuschel et al. 2006) seem to be popular approaches in the logic programming community and recent work has focused on scaling partial evaluation so that it can be applied to large programs (Leuschel and Vidal 2009; Vidal 2010). Our suggested usage of termination analysis in Section 4.8.1.2 is inspired by this line of work.

6.5 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 intended for the optimizer and making it harder to read and understand for humans.

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:

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

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

$$\mathit{foldr}\; f\; c\; (\mathit{build}\; g) \;=\; g\; f\; c$$

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 currently contains a variant of the short cut deforestation implemented using rewrite rules (Peyton Jones et al. 2001).

Takano and Meijer (1995) generalized short cut deforestation to work for any algebraic datatype through the acid rain theorem. Ghani and Johann (2008) have also generalized the *foldr/build* rule to a *fold/superbuild* rule that can eliminate intermediate structures of inductive types without disturbing the contexts in which they are situated.

Launchbury and Sheard (1995) worked on automatically transforming programs into suitable form for shortcut deforestation. Onoue et al. (1997) showed an implementation of the acid rain theorem for Gofer where they could automatically transform recursive functions into a form suitable for shortcut fusion. This allows the programmer to focus on writing beautiful programs that are easy to read without having to worry whether they are in a suitable form for the optimizer or not. Unfortunately we have not been able to find any further reports of this line of work.

6.6 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 also able to deduce completeness of the list abstraction algorithm. This completeness guarantees that if a list can be abstracted from a producer by abstracting its list constructors, then the list abstraction algorithm will do so.

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

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.

6.7 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*.

In more recent work Coutts et al. (2007) have extended these techniques to work on functions that handle nested lists, list comprehensions and filter-like functions. The main idea is to view structures as possibly finite streams and introduce a third option called skip for the stream producer. This coding makes it possible to fuse functions whose output lists are not necessarily of the same length as their input, such as *filter*.

6.8 Lightweight Fusion

Ohori's and Sasano's Lightweight Fusion (2007) works by promoting functions through the fix-point operator and guarantees termination by limiting inlining to at most once per function. They implement their transformation in a compiler for a variant of Standard ML and present some benchmarks. The algorithm is proven correct for a call-by-name language. It is explicitly mentioned that their goal is to extend the transformation to work for an impure call-by-value functional language.

Comparing lightweight fusion to our positive supercompiler is somewhat difficult, the algorithms themselves are not very similar. Comparing results of the algorithms is more straightforward – the restriction to only inline functions once makes lightweight fusion unable to handle successive applications of the same function or mutually recursive functions, something a positive supercompiler handles gracefully.

Ohori and Sasano show an example of how their method can fuse the intermediate list in $sum\ [1..1000]$, whereas our positive supercompiler will create some extra unwanted residual code from unfolding function definitions before generalizing the expression and fusing the intermediate lists. If we instead start with the expression $sum\ [x..y]$ both lightweight fusion and our algorithm will transform the expression to the same result. This also gives a hint about what the desired behaviour should be in this case: the numeric constants 1 and 1000 should be generalized before starting to supercompile the expression.

Finding out what expressions to generalize and when to generalize them appears to be tricky. Romanenko (1990) presents an arity raiser that will increase the number of parameters to functions under certain conditions. The work uses two analyses, a forward analysis to find opportunities for applying the transformation and a backward analysis for detecting the usefulness of the transformation. When applied before other transformations it was found to improve the structure of residual programs. A second approach that might be worth investigating is to base the decision of when to generalize on some kind of adjusted dependency analysis along the same lines as the work byAbadi and Lévy (1996).

For the particular example of having a closed and terminating expression to start with we could disable the whistle in our supercompiler and it would replace the starting expression with the computed final result, or we might refine the whistle to also account for integers in the same spirit as Albert et al. (2009).

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

6.9 Summary

6.9.1 Call-By-Value versus Call-By-Name

Deforestation, Supercompilation and GPC are all transformations that traditionally have call-by-name semantics. While the general supercompilation algorithm presented in Chapter 2.2 also have call-by-name semantics, we also presented a modification to the algorithm in Section 2.3 that allows it to preserve semantics of programs written in call-by-value languages.

It is well known that there are programs that will evaluate to a value under call-by-name, but not terminate at all under call-by-value, and we gave such an example in Chapter 1. We also know that the halting problem is undecidable, so it appears reasonable to expect that in order to preserve call-by-value semantics the transformation will be weaker in some sense. Despite this we have shown that our supercompiler actually does remove intermediate structures in many of the examples that were designed to show how useful particular call-by-name transformations are.

We also showed an example in Section 4.8.1 where our call-by-value algorithm does remove the intermediate structure, despite a call-by-name algorithm doing so. We managed to overcome that particular problem by strengthening the basic call-by-value algorithm, but the general problem remains: you can always construct an example where a call-by-value algorithm is not allowed to remove an expression because it might transform a non-terminating program into a terminating one.

With all that said, we also note that non-terminating expressions are not very interesting in practice since real programs that people use will contain very few of them.

Table 6.1: 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 CBV SCP (Section 2.3)	Unification	CBN	Yes
Positive CBV SCP (Section 4.8.1)	Unification	CBN	Yes
Positive SCP	Unification	CBN	Yes
Perfect SCP	Constraint	CBN	Yes
GPC	Constraint	CBN	Yes

6.9.2 Taxonomy of Transformers

Important characteristics of the positive supercompilers, using the same criteria as Glück and Sørensen (1996):

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 that preserves call-by-value semantics.

Our positive supercompiler propagates positive information into the branches of case-expressions, has an evaluation strategy that is call-by-name when it is safe to do so, and does remove intermediate data structures.

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 6.1.

6.9.3 Closest relative

The positive supercompiler (Sørensen et al. 1996) is the work which is closest to our work, especially if looking in Table 6.1. 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. The work on supercompiling call-by-need languages by Bolingbroke and Peyton Jones (2010) is also close to our work, especially our strengthened call-by-value algorithm from Section 4.8.1.

CHAPTER 7

Conclusions and Future Research

7.1 Conclusions

We have presented positive supercompilation algorithms for higher-order call-by-value and call-by-name languages that preserves termination properties of programs they optimize. We have proven the call-by-value algorithm correct and compared it with existing call-by-name transformations. It is clear from the examples and measurements that deforestation-like transformations are both possible and useful for call-by-value languages, as well as call-by-name languages.

We have also shown how to speculatively supercompile expressions and discard bad results, and implemented our call-by-name algorithm in GHC to perform measurements on the standard nofib benchmark suite. We managed to supercompile large parts of the imaginary and spectral parts of nofib in a matter of seconds while keeping the binary size increase below 5%.

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. While there are still quite a few open questions about speculatively supercompiling expressions and discarding bad results, mainly in how to decide what expressions to discard, our simple approach gave results that were a lot better than we expected.

7.2 Summary of Contributions

- The most important piece of this work is the positive supercompilation algorithm for higher-order languages, including folding, outlined in Chapter 2.

- We proved the call-by-value version of the algorithm correct, both termination and preservation of semantics in Chapter 3.

- We extend the rules of the algorithm with pre- and post-conditions that mitigate the problem of code explosion in Chapter 4.

- We provide a stronger algorithm for positive supercompilation for a strict and pure functional language (Section 4.8.1).
- We extend the supercompiler with a termination test that enables some unused let-expressions to be removed even though they are not fully evaluated. This feature is particularly beneficial in conjunction with the strengthened supercompilation algorithm in Section 4.8.1, since pushing bindings into case branches tend to result in many seemingly redundant let-expressions (Section 4.8.1.2).
- We extend the folding mechanism of the positive supercompiler to also perform the static argument transformation (Section 4.8.2).
- We present an alternative approach to avoid exponential code growth (Section 4.8.3).
- We extend our supercompiler to work on a limited impure language (Section 4.8.4).
- We summarize lessons learned during the implementation, and present preliminary measurements on two prototype implementations in Chapter 5.

7.3 Future Research

Supercompilation is still a rather expensive optimization, but it can undoubtedly produce great results on high level Haskell programs. The major obstacle we see at the moment is that supercompilation might prevent the compiler from applying other optimizations, which in turn can increase the runtime of the final executable by several orders of magnitude. There are several possible ways to tackle this problem:

- Integrate other optimizations into the supercompiler
- Make the supercompiler output programs that are friendly to other optimizations
- Strengthen other optimizations so that they can optimize the output from the supercompiler

The approximation of boring expressions could be extended in several ways to avoid transforming expressions that will give bad results. One observation is that the first call to reverse with an accumulating parameter ($reverse\ xs$ []) is not boring since it has a known constructor in the parameter, but by inspecting the body of $reverse$ it is obvious that this parameter is never used. One idea is to extend boring expressions to utilize strictness information so that it would discard this example because the static information present in the context is never used.

There is certainly room to improve the decision whether to discard supercompiled expressions. The constants currently used could be tuned further, and there is also the possibility to use other information that is present in the program.

A route that we have not explored is to use different acceptable-predicates for different parts of the supercompiler. Rule R18 and R19 could be extended with a similar test which could allow that only some parts of the surrounding context $\mathcal{R}$ is pushed down.

We observed that there are sometimes several candidates in the memoization list that the supercompiler could generalize the current expression against. Our current implementation selects the first one which is not necessarily the best candidate. A ranking function between different candidates would be useful.

Bolingbroke and Peyton Jones (2010) have presented a call-by-need algorithm for supercompilation. We believe that our current formulation of the supercompilation algorithm can be adapted to a call-by-need semantics, and knowing that the supercompiler does not duplicate computations would be helpful when investigating why some programs take a performance hit from supercompilation.

Leuschel (2002) has investigated interesting variations of the homeomorphic embedding that we believe are compatible with our design choices and could be adapted to work on our zipper representation.

The stronger supercompilation algorithm presented in Section 4.8.1 in its current formulation will traverse expressions repeatedly, which is a source of inefficiency. The reason to these retraversals is to detect when variables become strict and linear during the transformation. It might be possible to avoid these traversals by caching some information about use sites, or reduce the number of traversals by maintaining some kind of mapping from variable names to their binding locations.

We have provided a supercompilation algorithm for a higher-order call-by-value language, which covers the first semantic layer of Timber (Carlsson et al. 2003). Merging objects on the second semantic layer is a potential direction for future work. The benefits of automatically removing objects would be similar to the benefits at the expression layer: the programmer is allowed to spend his effort on correctly modelling the system and solving the problem at hand instead of writing low level code that runs fast. Liu et al. (2005) has done some work that at a high level appears to be similar and could perhaps serve as a starting point.

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, linearity, 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 and 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).

7.3.1 Termination Analysis

We suggested using termination analysis in Section 4.8.1.2 in order to remove list allocations and traversals that were only there to ensure that the supercompiler had preserved the program semantics. A different use of termination analysis suggested by Dave Sands would be to perform a termination analysis on the input program, and in that case use a call-by-name supercompilation algorithm to transform it. This would avoid the problem of having to repeatedly perform termination analysis on expressions throughout the transformation. Using a call-by-name algorithm for the transformation would also avoid the imprecision in the strictness analysis beyond the potential performance improvements of the supercompilation algorithm.

7.3.2 Variable Folding

We can extend our algorithm with an additional rule that tries to fold against expressions that are equivalent up to alpha renaming when the expression in focus is a variable. Adding a rule $\mathcal{D}[\![x]\!]\,\rho\,\mathcal{R}\,\sigma\,s = \mathcal{D}_{varapp}(x)\,\rho\,\mathcal{R}\,\sigma\,s$ right after rule R3 would be the first part. The definition of $\mathcal{D}_{varapp}()\ \rho\ \sigma$ should be similar to $\mathcal{D}_{app}()$, except that it does not need to invoke the whistle. The extension is beneficial since it reduces code size in general because more expressions will fold. Given that there is a variable currently in focus for the algorithm we therefore dub this variable folding, and introduce a new function $\mathcal{D}_{varapp}()$ as shown in Figure 7.1. An example of when this is useful is when supercompiling the right hand side of append in a module:

$$
\begin{array}{l}
append\ xs\ ys\ = \mathcal{D}[\![\ \textbf{case}\ xs\ \ \textbf{of}\,\{\ldots\}\ ]\!] \\[1em]
\{\ \textit{Create a fresh function and transform its body}\ \} \\[1em]
\begin{array}{lcl}
append\ xs\ ys & = & h\ xs\ ys \\
h\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [\,] \ \rightarrow\ ys \\
 & & \quad (x' : xs') \ \rightarrow\ x' : \mathcal{D}[\![\ append\ xs'\ ys]\!]
\end{array} \\[1em]
\{\ \textit{Inline append}\ \} \\[1em]
\begin{array}{lcl}
append\ xs\ ys & = & h\ xs\ ys \\
h\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [\,] \ \rightarrow\ ys \\
 & & \quad (x' : xs') \ \rightarrow\ x' : \mathcal{D}[\![\ \textbf{case}\ xs'\ \ \textbf{of}\,\{\ldots\}]\!]
\end{array} \\[1em]
\{\ \textit{Fold against h}\ \} \\[1em]
\begin{array}{lcl}
append\ xs\ ys & = & h\ xs\ ys \\
h\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [\,] \ \rightarrow\ ys \\
 & & \quad (x' : xs') \ \rightarrow\ x' : h\ xs'\ ys
\end{array}
\end{array}
$$

Without variable folding this example would give one more unrolling of the recursion which would have increased the code size.

The extension is easy to implement, but there is a proof technical problem since this extension introduces recursion into the program at points where the input program did not contain any recursion. Proving this extension correct is an interesting direction for our future work.

$$
\begin{array}{llll}
\mathcal{D}_{varapp}(x)\,\rho\,\mathcal{R}\,\sigma\,s & \mid \exists h.(\sigma \cup \rho)(h) \equiv \lambda\overline{x}.e_2 & = (h\,\overline{x}, \sigma, !s) & (5)\\
& \mid \mathit{acceptable}(\sigma_1 \backslash \sigma, [x]\mathcal{R}, e', s_1) & = (h\,\overline{x}, ((h, \overline{x}, e_2), e') : \sigma_1, s_1) & (6)\\
& \mid \text{otherwise} & = \mathcal{B}[\![x]\!]\,\rho\,\mathcal{R}\,\sigma\,s & (7)
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & (e', \sigma_1, s_1) = \mathcal{B}[\![x]\!]\,\rho'\,\mathcal{R}\,\sigma\,s\\
& e_2 = \mathit{normalize}([x]\mathcal{R})\\
& \rho' = (h, \lambda\overline{x}.e_2) : \rho\\
& h \text{ fresh}\\
& \overline{x} = \mathit{fv}(e_2)
\end{array}
$$

Figure 7.1: Folding on variables

7.3.3 Parallelism

Section 5.1 mentioned that some parts of the supercompiler are inherently parallel. The suggestions were the testing for folding and non-termination performed in $\mathcal{D}_{app}()$, but this can be extended further.

The current algorithm formulation is inherently sequential since the store σ is threaded through and potentially modified in all branches, but there might be a possibility to avoid threading the store by having local stores for each thread working in parallel. All right hand sides with several recursive calls in the driving and building algorithm could then be run in parallel, at the potential cost of extra work performed by the supercompiler and code duplication in the output program. However, merging two stores should be quite cheap since it is sufficient to compare the elements of the stores up to alpha renaming, and in that scenario discard one of these members and replace calls to that member with calls to the other member throughout the program.

We find it hard to speculate whether this will turn out to be a performance benefit in practice since it both requires some more post-processing and also relies on the parallel instances of the supercompiler being spawned at the right program points.

APPENDIX A

Function Definitions

```
square x        = x ∗ x

map f xs        = case xs of
                    [] → ys
                    (x′ : xs′) → f x′ : map f xs′
sum xs          = case xs of
                    [] → 0
                    (x′ : xs′) → x′ + sum xs′
zip xs ys       = case xs of
                    [] → []
                    (x′ : xs′) → case ys of
                                   [] → []
                                   (y′ : ys′) → (x′, y′) : zip xs′ ys′
append xs ys    = case xs of
                    [] → ys
                    (x′ : xs′) → x′ : append xs′ ys
foldr f c xs    = case xs of
                    [] → c
                    (x′ : xs′) → f x′ (foldr f c xs′)
fac n           = case n of
                    0 → 1
                    _ → n ∗ fac (n − 1)
const x y       = x

rev xs ys       = case xs of
                    [] → ys
                    (x′ : xs′) → rev xs′ (x′ : ys)
```

$$
\begin{array}{lll}
\mathit{zipWith}\ f\ \mathit{xs}\ \mathit{ys} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & & \quad [\,] \ \rightarrow\ [\,] \\
 & & \quad (x' : \mathit{xs}') \ \rightarrow\ \textbf{case}\ \mathit{ys}\ \textbf{of} \\
 & & \qquad\qquad\qquad [\,] \ \rightarrow\ [\,] \\
 & & \qquad\qquad\qquad (y' : \mathit{ys}') \rightarrow\ f\ x'\ y' : \mathit{zipWith}\ f\ \mathit{xs}'\ \mathit{ys}' \\
\mathit{fst}\ p & = & \textbf{case}\ p\ \textbf{of} \\
 & & \quad (x,\ y) \ \rightarrow\ x \\
\mathit{snd}\ p & = & \textbf{case}\ p\ \textbf{of} \\
 & & \quad (x,\ y) \ \rightarrow\ y
\end{array}
$$

APPENDIX B

Proofs

The complete proof for total correctness of the driving algorithm presented in Chapter 2 is presented.

B.1 Proof of Lemma 3.18

For each rule $\mathcal{D}[\![e_1]\!]\,\rho\,\mathcal{R}_1 = e'_1$ and $\mathcal{B}[\![e_2]\!]\,\rho\,\mathcal{R}_2 = e'_2$ in Definition 2.6 and 2.7 and each call $\mathcal{D}[\![e_3]\!]\,\rho'\,\mathcal{R}_3$ or $\mathcal{B}[\![e_4]\!]\,\rho\,\mathcal{R}_4$ in e'_1 or e'_2, $|\mathcal{D}[\![e_3]\!]\,\rho'\,\mathcal{R}_3| < |\mathcal{D}[\![e_1]\!]\,\rho\,\mathcal{R}_1|$, $|\mathcal{B}[\![e_4]\!]\,\rho\,\mathcal{R}_4| < |\mathcal{D}[\![e_1]\!]\,\rho\,\mathcal{R}_1|$, $|\mathcal{D}[\![e_3]\!]\,\rho'\,\mathcal{R}_3| < |\mathcal{B}[\![e_2]\!]\,\rho\,\mathcal{R}_2|$, and $|\mathcal{B}[\![e_4]\!]\,\rho\,\mathcal{R}_4| < |\mathcal{B}[\![e_2]\!]\,\rho\,\mathcal{R}_2|$.

Proof. Inspection of rules:

R1 We have LHS $\mathcal{D}[\![n_j]\!]\,\rho\,(\textbf{case}\ \square\ \textbf{of}\,\{n_i \to e_i\} : \mathcal{R})$ and the subproblem $\mathcal{D}[\![e_j]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![n_j]\!]\,\rho\,(\textbf{case}\ \square\ \textbf{of}\,\{n_i \to e_i\} : \mathcal{R})| = (\rho, M, |[\textbf{case}\, n_j\, \textbf{of}\,\{n_i \to e_i\}]\mathcal{R}|, A, |n_j|)$ and $|\mathcal{D}[\![e_j]\!]\,\rho\,\mathcal{R}| = (\rho, M', |[e_j]\mathcal{R}|, A', |e_j|)$. We know that $M' < M$ since the transformation eliminated one case reduction.

R2 We have LHS $\mathcal{D}[\![n]\!]\,\rho\,(n_1 \oplus \square : \mathcal{R})$ and the subproblem $\mathcal{D}[\![n_2]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![n]\!]\,\rho\,(n_1 \oplus \square : \mathcal{R})| = (\rho, M, |[n_1 \oplus n]\mathcal{R}|, A, |n|)$ and $|\mathcal{D}[\![n_2]\!]\,\rho\,\mathcal{R}| = (\rho, M', |[n_2]\mathcal{R}|, A', |n_2|)$. No case-expressions are eliminated in the transformation so $M = M'$. It is clear that $|[n_2]\mathcal{R}| < |[n_1 \oplus n]\mathcal{R}|$ since $|n_2| < |n_1 \oplus n|$.

R3 Inline the definition of $\mathcal{D}_{app}(\,)$:

(1) We have LHS $\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R}$ and no subproblems in the RHS, so Lemma 3.18 holds vacuously.

(2) We have LHS $\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R}$ with subproblems $\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon$ and $\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon$ where $\overline{f}$ and f_g are constructed by splitting $[g]\mathcal{R}$. $|\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[g]\mathcal{R}|, A, |g|)$, $|\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon| = (\rho, M', |\overline{f}|, A', \overline{f})$ and $|\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon| = (\rho, M'', |f_g|, A'', |f_g|)$. We know that $M' \leq M$ and $M'' \leq M$ because $\overline{f}$ and f_g are proper subexpressions of $[g]\mathcal{R}$ as well as $|f_g| < |[g]\mathcal{R}|$ and $\forall f \in \overline{f}.|f| < |[g]\mathcal{R}|$ for the same reason.

(3) We have LHS $\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R}$ with subproblem $\mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R}$. $|\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[g]\mathcal{R}|, A, |g|)$ and $|\mathcal{D}[\![e]\!]\,\rho'\,\mathcal{R}| = (\rho', M', |[e]\mathcal{R}|, A', |e|)$. We know that ρ is a suffix of ρ', so the weight is decreased.

R4 We have LHS $\mathcal{D}[\![k_j]\!]\,\rho\,(\overline{\Box\ e} : \mathbf{case}\ \Box\ \ \mathbf{of}\,\{k_i\,\overline{x}_i\ \rightarrow\ e_i\ \} : \mathcal{R})$ and the subproblem $\mathcal{D}[\![\mathbf{let}\,\overline{x}_j = \overline{e}\,\mathbf{in}\,e_j]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![k_j]\!]\,\rho\,(\overline{\Box\,e} : \mathbf{case}\ \Box\ \ \mathbf{of}\,\{k_i\,\overline{x}_i \rightarrow e_i\} : \mathcal{R})| = (\rho, M,$ $|[\mathbf{case}\,k_j\,\overline{e}\,\mathbf{of}\,\{k_i\,\overline{x}_i \rightarrow e_i\}]\mathcal{R}|, A', |k_j|)$ and $|\mathcal{D}[\![\mathbf{let}\,\overline{x}_j = \overline{e}\,\mathbf{in}\,e_j]\!]\,\rho\,\mathcal{R}| = (\rho, M',$ $|[\mathbf{let}\,\overline{x}_j = \overline{e}\,\mathbf{in}\,e_j]\mathcal{R}|, A', |\mathbf{let}\,\overline{x}_j\ =\ \overline{e}\,\mathbf{in}\,e_j|)$. We know that $M' < M$ since the transformation eliminated one case reduction.

R5 We have LHS $\mathcal{D}[\![\lambda\,\overline{x}.e_1]\!]\,\rho\,(\overline{\Box\ e_2} : \mathcal{R})$ and the subproblem $\mathcal{D}[\![\mathbf{let}\ \overline{x}\ =\ \overline{e_2}\ \mathbf{in}\ e_1]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![\lambda\overline{x}.e_1]\!]\,\rho\,(\overline{\Box\ e_2} : \mathcal{R})| = (\rho, M, |[(\lambda\overline{x}.e_1)\,\overline{e_2}]\mathcal{R}|, A, |\lambda\overline{x}.e_1|)$ and $|\mathcal{D}[\![\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\!]\,\rho\,\mathcal{R}| =$ $(\rho, M', |[\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\mathcal{R}|, A', |\mathbf{let}\,\overline{x}\ =\ \overline{e_2}\,\mathbf{in}\,e_1|)$. No case-expressions are removed or introduced, so $M' = M$. Assume $\overline{e_2}$ consists of k expressions. We need to show that $|[(\lambda\overline{x}.e_1)\,\overline{e_2}]\mathcal{R}|$ $> |[\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\mathcal{R}|$. We can eliminate $\mathcal{R}$ from both sides and expand the weight of the expressions according to the definition, leaving to show that $k + |e_1| + k + |\overline{e_2}| > k + |\overline{e_2}| + |e_1|$. This holds because $k > 0$ and $2k > k$.

R6 We have LHS $\mathcal{D}[\![\lambda x.e]\!]\,\rho\,\mathcal{R}$ and subproblems $e' = \mathcal{D}[\![e]\!]\,\rho\,\epsilon$ and $\mathcal{B}[\![\lambda x.e']\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![\lambda x.e]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[\lambda x.e]\mathcal{R}|, A, |\lambda x.e|)$, $|\mathcal{D}[\![e]\!]\,\rho\,\epsilon| = (\rho, M', |e|, A', |e|)$ and $|\mathcal{B}[\![\lambda x.e']\!]\,\rho\,\mathcal{R}| = (\rho, M'', |\mathcal{R}|, A'', 0)$. It is clear that $M' \leq M$ and $|e| < |[\lambda x.e]\mathcal{R}|$ since *e* is a proper subexpression of $[\lambda x.e]\mathcal{R}$, and that $M'' \leq M$ and $|\mathcal{R}| < |[\lambda x.e]\mathcal{R}|$ because $\mathcal{R}$ is a proper subexpression of the latter.

R7 We have LHS $\mathcal{D}[\![\mathbf{let}\,x = n\,\mathbf{in}\,e]\!]\,\rho\,\mathcal{R}$ and the subproblem $\mathcal{D}[\![[n/x]e]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![\mathbf{let}\,x = n\,\mathbf{in}\,e]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[\mathbf{let}\,x = n\,\mathbf{in}\,e]\mathcal{R}|, A, |\mathbf{let}\,x = n\,\mathbf{in}\,e|)$ and $|\mathcal{D}[\![[n/x]e]\!]\,\rho\,\mathcal{R}| =$ $(\rho, M', |[[n/x]e]\mathcal{R}|, A', |[n/x]e|)$. Since no case-expressions are removed or introduced by this step, $M = M'$. Also, $|[[n/x]e]\mathcal{R}| < |[\mathbf{let}\,x = n\,\mathbf{in}\,e]\mathcal{R}|$ since the transformation removes a let-expression,

R8 We have LHS $\mathcal{D}[\![\mathbf{let}\,x = y\,\mathbf{in}\,e]\!]\,\rho\,\mathcal{R}$ and the subproblem $\mathcal{D}[\![[y/x]e]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![\mathbf{let}\,x = y\,\mathbf{in}\,e]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[\mathbf{let}\,x = y\,\mathbf{in}\,e]\mathcal{R}|, A, |\mathbf{let}\,x = y\,\mathbf{in}\,e|)$ and $|\mathcal{D}[\![[y/x]e]\!]\,\rho\,\mathcal{R}| =$ $(\rho, M', |[[y/x]e]\mathcal{R}|, A', |[y/x]e|)$ which is smaller since *e* is a proper subexpression of $\mathbf{let}\,x =$ $y\,\mathbf{in}\,e$ and therefore $M' \leq M$ and $|[[y/x]e]\mathcal{R}| < |[\mathbf{let}\,x = y\,\mathbf{in}\,e]\mathcal{R}|$.

R9 We have LHS $\mathcal{D}[\![\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\!]\,\rho\,\mathcal{R}$ and the subproblems $\mathcal{D}[\![[e_1/x]\,e_2]\!]\,\rho\,\mathcal{R}$ if *x* is linear in e_2, or $\mathcal{D}[\![e_1]\!]\,\rho\,\epsilon$ and $\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R}$ otherwise. $|\mathcal{D}[\![\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\!]\,\rho\,\mathcal{R}| = (\rho, M,$ $|[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R}|, A, |\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2|)$, $|\mathcal{D}[\![[e_1/x]e_2]\!]\,\rho\,\mathcal{R}| = (\rho, M', |[[e_1/x]e_2]\mathcal{R}|, A',$ $|[e_1/x]e_2|)$, $|\mathcal{D}[\![e_1]\!]\,\rho\,\epsilon| = (\rho, M'', |e_1|, A'', |e_1|)$ and $|\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R}| = (\rho, M'', |[e_2]\mathcal{R}|, A''', |e_2|)$. We know that $M' \leq M$, $M'' \leq M$ and that $|e_1| < |[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R}|$ as well as $|[e_2]\mathcal{R}| <$ $|[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R}|$ because e_1 and $[e_2]\mathcal{R}$ are proper subexpressions of $[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R}$.

R10 We have LHS $\mathcal{D}[\![n]\!]\,\rho\,(\Box \oplus e_2 : \mathcal{R})$ and the subproblem $\mathcal{D}[\![e_2]\!]\,\rho\,(n\ \oplus\ \Box : \mathcal{R})$. $|\mathcal{D}[\![n]\!]\,\rho\,(\Box \oplus e_2 : \mathcal{R})| = (\rho, M, |[n\ \oplus\ e_2]\mathcal{R}|, A, |n|)$ and $|\mathcal{D}[\![e_2]\!]\,\rho\,(n\ \oplus \Box : \mathcal{R})| = (\rho, M',$ $|[n\ \oplus\ e_2]\mathcal{R}|, A', |e_2|)$. It is clear that $M' = M$ since the transformation does not introduce or remove any case-expressions. It is also clear that $|[n\ \oplus\ e_2]\mathcal{R}| = |[n\ \oplus\ e_2]\mathcal{R}|$ since the expressions are syntactically equivalent. The transformation removed one left frame from the context, making $A' < A$ so the weight is reduced.

R11 We have LHS $\mathcal{D}[\![e_1 \oplus e_2]\!]\,\rho\,\mathcal{R}$ and the subproblem $\mathcal{D}[\![e_1]\!]\,\rho\,(\Box \oplus e_2 : \mathcal{R})$. $|\mathcal{D}[\![e_1 \oplus e_2]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[e_1 \oplus e_2]\mathcal{R}|, A, |e_1 \oplus e_2|)$ and $|\mathcal{D}[\![e_1]\!]\,\rho\,(\Box \oplus e_2 : \mathcal{R})| = (\rho, M',$ $|[e_1 \oplus e_2]\mathcal{R}|, A', |e_1|)$. It is clear that $M = M'$ and $|[e_1 \oplus e_2]\mathcal{R}| = |[e_1 \oplus e_2]\mathcal{R}|$ since the terms are equivalent. Also, $A = A'$ since there is one left frame introduced into the context and one arithmetic expression removed from $e_1 \oplus e_2$ in *A'*. However, $|e_1| < |e_1 \oplus e_2|$ since e_1 is a proper subexpression of the latter.

R12 We have LHS $\mathcal{D}[\![e_1\,e_2]\!]\,\rho\,\mathcal{R}$ and the subproblem $\mathcal{D}[\![e_1]\!]\,\rho\,(\Box\; e_2 : \mathcal{R})$. $|\mathcal{D}[\![e_1\,e_2]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[e_1\,e_2]\mathcal{R}|, A, |e_1\,e_2|)$ and $|\mathcal{D}[\![e_1]\!]\,\rho\,(\Box\; e_2 : \mathcal{R})| = (\rho, M', |[e_1\,e_2]\mathcal{R}|, A', |e_1|)$. It is clear that $M = M'$ and $|[e_1\,e_2]\mathcal{R}| = |[e_1\,e_2]\mathcal{R}|$ since the expressions are syntactically equivalent. Also, $A = A'$ since the number of arithmetic expressions and left frames remain unchanged. It is also clear that $|e_1| < |e_1\,e_2|$ since e_1 is a proper subexpression of the latter.

R13 We have LHS $\mathcal{D}[\![\mathbf{case}\, e\ \mathbf{of}\,\{p_i \to e_i\}]\!]\,\rho\,\mathcal{R}$ and the subproblem $\mathcal{D}[\![e]\!]\,\rho\,(\mathbf{case}\ \Box\ \mathbf{of}\,\{p_i \to e_i\} : \mathcal{R})$. $|\mathcal{D}[\![\mathbf{case}\, e\ \mathbf{of}\,\{p_i \to e_i\}]\!]\,\rho\,\mathcal{R}| = (\rho, M,$ $|[\mathbf{case}\, e\ \mathbf{of}\,\{p_i \to e_i\}]\mathcal{R}|, A, |\mathbf{case}\, e\ \mathbf{of}\,\{p_i \to e_i\}|)$ and $|\mathcal{D}[\![e]\!]\,\rho\,(\mathbf{case}\ \Box\ \mathbf{of}\,\{p_i \to e_i\} : \mathcal{R})| =$ $(\rho, M', |[\mathbf{case}\, e\,\mathbf{of}\,\{p_i \to e_i\}]\mathcal{R}|, A', |e|)$. It is clear that $M = M'$ and $|[\mathbf{case}\, e\,\mathbf{of}\,\{p_i \to e_i\}]\mathcal{R}| =$ $|[\mathbf{case}\, e\,\mathbf{of}\,\{p_i \to e_i\}]\mathcal{R}|$ since the expressions are syntactically equivalent. Also, $A = A'$ since the number of arithmetic expressions and left frames remain unchanged. It is also clear that $|e| < |\mathbf{case}\, e\ \mathbf{of}\,\{p_i \to e_i\}|$ since e is a proper subexpression of the latter.

R14 We have LHS $\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}$ and the subproblem $\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}$. $|\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}| = (\rho, M, |[e]\mathcal{R}|, A, |e|)$ and $|\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}| = (\rho, M', |\mathcal{R}|, A', 0)$. Since no expressions are eliminated by this step in the transformation, $M = M'$. Clearly $|\mathcal{R}| < |[e]\mathcal{R}|$ since $\mathcal{R}$ is a proper subexpression of the latter.

R15 We have LHS $\mathcal{B}[\![e']\!]\,\rho\,(\Box \oplus e_2 : \mathcal{R})$ and subproblems $\mathcal{D}[\![e_2]\!]\,\rho\,\epsilon$ and $\mathcal{B}[\![e' \oplus e_2']\!]\,\rho\,\mathcal{R}$. $|\mathcal{B}[\![e']\!]\,\rho\,(\Box \oplus e_2 : \mathcal{R})| = (\rho, M, |\,\Box \oplus e_2 : \mathcal{R}|, A, 0)$, $|\mathcal{D}[\![e_2]\!]\,\rho\,\epsilon| = (\rho, M', |e_2|, A', |e_2|)$ and $|\mathcal{B}[\![e' \oplus e_2']\!]\,\rho\,\mathcal{R}| = (\rho, M'', |\mathcal{R}|, A'', 0)$. We know that $M' \le M$ and $M'' \le M$ since e_2 and $\mathcal{R}$ are proper subexpressions of $[e' \oplus e_2]\mathcal{R}$, and that $|e_2| < |\,\Box \oplus e_2 : \mathcal{R}|$ as well as $|\mathcal{R}| < |\,\Box \oplus e_2 : \mathcal{R}|$ for the same reason.

R16 We have LHS $\mathcal{B}[\![e']\!]\,\rho\,(e_1' \oplus \Box : \mathcal{R})$ and the subproblem $\mathcal{B}[\![e_1' \oplus e']\!]\,\rho\,\mathcal{R}$. $|\mathcal{B}[\![e']\!]\,\rho\,(e_1' \oplus \Box : \mathcal{R})| = (\rho, M, |e_1' \oplus \Box : \mathcal{R}|, A, 0)$ and $|\mathcal{B}[\![e_1' \oplus e']\!]\,\rho\,\mathcal{R}| = (\rho, M', |\mathcal{R}|, A', 0)$. We know that $M' = M$ since no case-expressions are eliminated in this step of the transformation, and clearly $|\mathcal{R}| < |e_1' \oplus \Box : \mathcal{R}|$ since $\mathcal{R}$ is a proper subexpression of the latter.

R17 We have LHS $\mathcal{B}[\![e']\!]\,\rho\,(\Box\, e : \mathcal{R})$ and subproblems $\mathcal{D}[\![e]\!]\,\rho\,\epsilon$ and $\mathcal{B}[\![e'e'']\!]\,\rho\,\mathcal{R}$. $|\mathcal{B}[\![e']\!]\,\rho\,(\Box\, e : \mathcal{R})| = (\rho, M, |\Box\, e : \mathcal{R}|, A, 0)$, $|\mathcal{D}[\![e]\!]\,\rho\,\epsilon| = (\rho, M', |e|, A', |e|)$ and$|\mathcal{B}[\![e'\,e'']\!]\,\rho\,\mathcal{R}|$ $= (\rho, M'', |\mathcal{R}|, A'', 0)$. We know that $M' \le M$ and $|e| < |\,\Box\ e : \mathcal{R}|$ because *e* is a proper subexpression of $[e'\,e]\mathcal{R}$. We also know that $M'' \le M$ and $|\mathcal{R}| < |\,\Box\ e : \mathcal{R}|$ since $\mathcal{R}$ is a proper subexpression of the latter.

R18 We have LHS $\mathcal{B}[\![x']\!]\,\rho\,(\mathbf{case}\ \Box\ \mathbf{of}\,\{p_i \to e_i\} : \mathcal{R})$ and the subproblems $\mathcal{D}[\![[p_i/x']e_i]\!]\,\rho\,([p_i/x']\mathcal{R})$. $|\mathcal{B}[\![x']\!]\,\rho\,(\mathbf{case}\ \Box\ \mathbf{of}\,\{p_i \to e_i\} : \mathcal{R})| = (\rho, M, |\mathbf{case}\ \Box\ \mathbf{of}\,\{p_i \to e_i\} : \mathcal{R}|, A, 0)$ and $|\mathcal{D}[\![[p_i/x']e_i]\!]\,\rho\,([p_i/x']\mathcal{R})| = (\rho, M', |[p_i/x'][e_i]\mathcal{R}|, A', |[p_i/x']e_i|)$. We know that $M' < M$ since one case expression counted by M is not counted by M', and that the pattern substitutions do not introduce or duplicate any case-expressions or left frames.

R19 We have LHS $\mathcal{B}[\![e']\!]\,\rho\,(\textbf{case}\ \Box\ \textbf{of}\,\{p_i \to e_i\} : \mathcal{R})$ with subproblems $\mathcal{D}[\![e_i]\!]\,\rho\,\mathcal{R}$. $|\mathcal{B}[\![e']\!]\,\rho\,(\textbf{case}\ \Box\ \textbf{of}\,\{p_i \to e_i\} : \mathcal{R})| = (\rho, M, |\textbf{case}\,\Box\,\textbf{of}\,\{p_i \to e_i\} : \mathcal{R}|, A, 0)$ and $|\mathcal{D}[\![e_i]\!]\,\rho\,\mathcal{R}|$ $= (\rho, M', |[e_i]\mathcal{R}|, A', |e_i|)$. We know that $M' < M$ since there is one case-expression less counted by M'.

R20 We have LHS $\mathcal{B}[\![e']\!]\,\rho\,\epsilon$ and no subproblems in the RHS, so Lemma 3.18 holds vacuously.

$\Box$

B.2 Proof of Supporting Lemmas

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

Lemma B.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 3.29 the initial expressions are cost equivalent. □

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

Proof (Similar to Sands (1996a)). By inspection of the rules for $\mathcal{D}[\![\;]\!]$, all free occurrences of *h* in $\mathcal{D}[\![v]\!]\,\rho'\,\mathcal{R}$ must occur in sub-expressions of the form $h\,\overline{x}$. Suppose there are *k* such occurrences, which we can write as $\theta_1 h\,\overline{x}\ldots\theta_k h\,\overline{x}$, where the θ_i are just renamings of the variables $\overline{x}$. So $\mathcal{D}[\![v]\!]\,\rho'\,\mathcal{R}$ can be written as $[\theta_1 h\,\overline{x}\ldots\theta_k h\,\overline{x}/z_1\ldots z_k]e'$, where e' contains no free occurrences of *h*. Then (substitution associates to the right):

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

□

Lemma B.3. $[\textbf{let}\, x = e\,\textbf{in}\, f]\mathcal{R} \trianglerighteq_s \textbf{let}\, x = e\,\textbf{in}\,[f]\mathcal{R}$

Proof. Notice that $[\textbf{let}\, x = \square\,\textbf{in}\, f]\mathcal{R}$ is a redex, and assume $e \mapsto^k v$. The LHS evaluates in k steps $[\textbf{let}\, x = e\,\textbf{in}\, f]\mathcal{R} \mapsto^k [\textbf{let}\, x = v\,\textbf{in}\, f]\mathcal{R} \mapsto [[v/x]f]\mathcal{R}$, and the RHS evaluates in k steps $\textbf{let}\, x = e\,\textbf{in}\,[f]\mathcal{R} \mapsto^k \textbf{let}\, x = v\,\textbf{in}\,[f]\mathcal{R} \mapsto [v/x][f]\mathcal{R}$. Since contexts do not bind variables these two terms are equivalent and by Lemma 3.29 the initial terms are cost equivalent. □

Lemma B.4. $[\textbf{letrec}\, g = v\,\textbf{in}\, e]\mathcal{R} \trianglerighteq_s \textbf{letrec}\, g = v\,\textbf{in}\,[e]\mathcal{R}$

Proof. Translate both sides according to the definition into $[(\lambda g.e)\,(\lambda n.fix\,(\lambda g.v)\,n)]\mathcal{R} \trianglerighteq_s (\lambda g.[e]\mathcal{R})\,(\lambda n.fix\,(\lambda g.v)\,n)$. Notice that $[\square]\mathcal{R}$ is a redex. The LHS evaluates in 0 steps $[(\lambda g.e)\,(\lambda n.fix\,(\lambda g.v)\,n)]\mathcal{R} \mapsto [[\lambda n.fix\,(\lambda g.v)\,n/g]e]\mathcal{R}$ and the RHS $(\lambda g.[e]\mathcal{R})\,(\lambda n.fix\,(\lambda g.v)\,n) \mapsto [\lambda n.fix\,(\lambda g.v)\,n/g][e]\mathcal{R}$. Since our contexts do not bind variables these two terms are equivalent and by Lemma 3.29 the initial terms are cost equivalent. □

Lemma B.5. $[\textbf{case}\, e\,\textbf{of}\,\{p_i \to e_i\}]\mathcal{R} \trianglerighteq_s \textbf{case}\, e\,\textbf{of}\,\{p_i \to [e_i]\mathcal{R}\}$

Proof. Notice that $[\textbf{case}\ \square\ \textbf{of}\,\{p_i \to e_i\}]\mathcal{R}$ is a redex, and assume $e \mapsto^k n_j$. The LHS evaluates in k steps $[\textbf{case}\, e\,\textbf{of}\,\{p_i \to e_i\}]\mathcal{R} \mapsto^k [\textbf{case}\, n_j\,\textbf{of}\,\{p_i \to e_i\}]\mathcal{R} \mapsto [e_j]\mathcal{R}$, and the RHS evaluates in k steps $\textbf{case}\, e\,\textbf{of}\,\{p_i \to [e_i]\mathcal{R}\} \mapsto^k \textbf{case}\, n_j\,\textbf{of}\,\{p_i \to [e_i]\mathcal{R}\}[f]\mathcal{R} \mapsto [e_j]\mathcal{R}$. Since these two terms are equivalent the initial terms are cost equivalent by Lemma 3.29.

□

B.3 Proof of Lemma 3.35

Let $[e]\mathcal{R}$ be an expression and ρ a memoization list such that

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

then $[e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R})$ if $[e']\mathcal{R}' \trianglerighteq_s \rho'(\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}')$ for all $e', \mathcal{R}', \rho'$ such that $|\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}'| < |\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}|$.

We reason by induction on the weight of the building problem $\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}$ by case analysis on the rule matching $\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}$.

B.3.1 R15

We have that $\rho(\mathcal{B}[\![e']\!]\,\rho\,(\square \oplus e_2 : \mathcal{R})) = \rho(\mathcal{B}[\![e' \oplus e_2']\!]\,\rho\,\mathcal{R})$ where $e_2' = \mathcal{D}[\![e_2]\!]\,\rho\,\epsilon$. By the induction hypothesis $[e' \oplus e_2']\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e' \oplus e_2']\!]\,\rho\,\mathcal{R})$. The conditions of the lemma ensure that $fv(e_2) \cap dom(\rho) = \emptyset$ and that $fv([e' \oplus e_2']\mathcal{R}) \cap dom(\rho) = \emptyset$ so $\rho(e_2) = e_2$ and $\rho([e' \oplus e_2']\mathcal{R}) = [e' \oplus e_2']\mathcal{R}$ and we can conclude from the assumption that $e_2 \trianglerighteq_s \rho(\mathcal{D}[\![e_2]\!]\,\rho\,\epsilon)$ since Lemma 3.18 guarantees $|\mathcal{D}[\![e_2]\!]\,\rho\,\epsilon| < |\mathcal{B}[\![e']\!]\,\rho\,(\square \oplus e_2 : \mathcal{R})|$. Together with congruence properties of improvement this allows us to conclude $[e' \oplus e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e' \oplus e_2']\!]\,\rho\,\mathcal{R})$.

B.3.2 R16

We have that $\rho(\mathcal{B}[\![e']\!]\,\rho\,(e_1' \oplus \square : \mathcal{R})) = \rho(\mathcal{B}[\![e_1' \oplus e']\!]\,\rho\,\mathcal{R})$. The conditions of the lemma ensures that $fv([e_1' \oplus e']\mathcal{R}) \cap dom(\rho) = \emptyset$ so $\rho([e_1' \oplus e']\mathcal{R}) = [e_1' \oplus e']\mathcal{R}$ and by the induction hypothesis $[e_1' \oplus e']\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e_1' \oplus e']\!]\,\rho\,\mathcal{R})$ which is syntactically equivalent to the input, and we conclude $[e_1' \oplus e']\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e']\!]\,\rho\,(e_1' \oplus \square : \mathcal{R}))$.

B.3.3 R17

We have that $\rho(\mathcal{B}[\![e']\!]\,\rho\,(\square\, e : \mathcal{R})) = \rho(\mathcal{B}[\![e'e'']\!]\,\rho\,\mathcal{R})$ where $e'' = \mathcal{D}[\![e]\!]\,\rho\,\epsilon$. The conditions of the lemma ensure that $fv(e) \cap dom(\rho) = \emptyset$ and that $fv([e'\,e]\mathcal{R}) \cap dom(\rho) = \emptyset$ so $\rho(e) = e$ and we can conclude from the the assumption that $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]\,\rho\,\epsilon)$ since Lemma 3.18 guarantees $|\mathcal{D}[\![e]\!]\,\rho\,\epsilon| < |\mathcal{B}[\![e']\!]\,\rho\,(\square\, e : \mathcal{R})|$. By the induction hypothesis $[e'\,e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e'e]\!]\,\rho\,\mathcal{R})$ which taken together with congruence properties of improvement allows us to conclude $[e'\,e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e']\!]\,\rho\,(\square\, e : \mathcal{R}))$.

B.3.4 R18

We have that $\rho(\mathcal{B}[\![x']\!]\,\rho\,(\textbf{case}\ \square\ \ \textbf{of}\,\{p_i \rightarrow e_i\} : \mathcal{R})) =$
$\rho(\textbf{case}\,x'\ \textbf{of}\,\{p_i \rightarrow \mathcal{D}[\![[p_i/x']e_i]\!]\,\rho\,([p_i/x']\mathcal{R})\})$. The condition of the lemma ensure that
$fv([\textbf{case}\,x'\ \textbf{of}\,\{p_i \rightarrow e_i\}]\mathcal{R}) \cap dom(\rho) = \emptyset$ so $\rho(\textbf{case}\,x'\ \textbf{of}\,\{p_i \rightarrow \mathcal{D}[\![[p_i/x']e_i]\!]\,\rho\,([p_i/x']\mathcal{R})\}) =$
$\textbf{case}\,x'\ \textbf{of}\,\{p_i \rightarrow \rho(\mathcal{D}[\![[p_i/x']e_i]\!]\,\rho\,([p_i/x']\mathcal{R}))\}$ and we can conclude from the assumption that $[p_i/x'][e_i]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![[p_i/x']e_i]\!]\,\rho\,([p_i/x']\mathcal{R})\})$ since Lemma 3.18 guarantees a smaller weight

of the driving problem. By Lemma B.5 the input is strongly improved by $\textbf{case}\, x'\, \textbf{of}\, \{p_i \rightarrow [e_i]\mathcal{R}\}$, and taken together with congruence properties of improvement this allows us to conclude $[\textbf{case}\, x'\, \textbf{of}\, \{p_i \rightarrow e_i\}]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![x']\!]\, \rho\, (\textbf{case}\ \square\ \ \textbf{of}\, \{p_i \rightarrow e_i\} : \mathcal{R}))$.

B.3.5 R19

Argument analogous to R18.

B.3.6 R20

We have that $\rho(\mathcal{B}[\![e']\!]\, \rho\, \epsilon) = \rho(e')$ and the conditions of the lemma ensure that $\mathit{fv}(e') \cap \mathit{dom}(\rho) = \emptyset$ so $\rho(e') = e'$. This is syntactically equivalent to the input and we conclude $e' \trianglerighteq_s \rho(\mathcal{B}[\![e']\!]\, \rho\, \epsilon)$.

B.4 Proof of Theorem 3.36

We set out to prove the theorem that if $[e]\mathcal{R}$ is an expression, and ρ a memoization list such that

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

then $[e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]\, \rho\, \mathcal{R})$. We reason by induction on the weight of the driving problem $\mathcal{D}[\![e]\!]\, \rho\, \mathcal{R}$ by case analysis on the rule matching $\mathcal{D}[\![e]\!]\, \rho\, \mathcal{R}$.

B.4.1 R1

We have that $\rho(\mathcal{D}[\![n_j]\!]\, \rho\, (\textbf{case}\ \square\ \ \textbf{of}\, \{n_i \rightarrow e_i\} : \mathcal{R})) = \rho(\mathcal{D}[\![e_j]\!]\, \rho\, \mathcal{R})$. By the induction hypothesis $[e_j]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e_j]\!]\, \rho\, \mathcal{R})$ and the conditions of the theorem ensure that
$\mathit{fv}([\textbf{case}\, n_j\, \textbf{of}\, \{n_i \rightarrow e_i\}]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so $\rho([e_j]\mathcal{R}) = [e_j]\mathcal{R}$. Since $[\textbf{case}\, n_j\, \textbf{of}\, \{n_i \rightarrow e_i\}]\mathcal{R} \mapsto [e_j]\mathcal{R}$ it follows that $[\textbf{case}\, n_j\, \textbf{of}\, \{n_i \rightarrow e_i\}]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![n_j]\!]\, \rho\, (\textbf{case}\ \square\ \ \textbf{of}\, \{n_i \rightarrow e_i\} : \mathcal{R}))$.

B.4.2 R2

We have that $\rho(\mathcal{D}[\![n]\!]\, \rho\, (n_1 \oplus \square : \mathcal{R})) = \rho(\mathcal{D}[\![n_2]\!]\, \rho\, \mathcal{R})$. By the induction hypothesis $[n_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![n_2]\!]\, \rho\, \mathcal{R})$ and the conditions of the theorem ensure that $\mathit{fv}([n_2]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so $\rho([n_2]\mathcal{R}) = [n_2]\mathcal{R}$. Since $[n_1 \oplus n]\mathcal{R} \mapsto [n_2]\mathcal{R}$ it follows that $[n_1 \oplus n]\mathcal{R} \trianglerighteq_s$
$\rho(\mathcal{D}[\![n]\!]\, \rho\, (n_1 \oplus \square : \mathcal{R}))$.

B.4.3 R3

B.4.3.1 Case: (1)

Suppose $\exists h.\rho|_g(h) \equiv \lambda \overline{x}.[g]\mathcal{R}$ and hence that $\mathcal{D}[\![g]\!]\, \rho\, \mathcal{R} = h\, \overline{x}$

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

B.4.3.2 Case: (2)

Suppose $\exists h.\rho|_g(h) \trianglelefteq \lambda\overline{x}.[g]\mathcal{R}$ **and hence that** $\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R} = [\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon/\overline{y}]\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon$

We have that $\rho(\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R}) = \rho([\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon/\overline{y}]\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon) = [\rho(\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon)/\overline{x}]\rho(\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon)$. By the induction hypothesis, $\overline{f} \trianglerighteq_s \rho(\mathcal{D}[\![\overline{f}]\!]\,\rho\,\epsilon)$ and $f_g \trianglerighteq_s \rho(\mathcal{D}[\![f_g]\!]\,\rho\,\epsilon)$ and by congruence properties of strong improvement (Lemma 3.31:1) $[g]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![g]\!]\,\rho\,\mathcal{R})$.

B.4.3.3 Case: (3)

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

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

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

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

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

By Theorem 3.34 it is sufficient to show:

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

By Lemma B.2 and $\mathbf{letrec}\,h = \lambda\overline{x}.[v]\mathcal{R}\,\mathbf{in}\,[v]\mathcal{R} \trianglelefteq\!\trianglerighteq [v]\mathcal{R}$, this is equivalent to showing that

$$[v]\mathcal{R} \trianglerighteq_s \rho'(\mathcal{D}[\![v]\!]\,\rho'\,\mathcal{R})$$

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

B.4.4 R4

We have that $\rho(\mathcal{D}[\![k_j]\!]\,\rho\,(\overline{\square\,e} : \mathbf{case}\ \square\ \ \mathbf{of}\,\{k_i\,\overline{x_i} \rightarrow e_i\} : \mathcal{R})) = \rho(\mathcal{D}[\![\mathbf{let}\,\overline{x_j} = \overline{e}\,\mathbf{in}\,e_j]\!]\,\rho\,\mathcal{R})$. Evaluating the input term yields $[\mathbf{case}\,k_j\,\overline{e}\,\mathbf{of}\,\{p_i \rightarrow e_i\}]\mathcal{R} \mapsto^r [\mathbf{case}\,k_j\,\overline{v}\,\mathbf{of}\,\{p_i \rightarrow e_i\}]\mathcal{R} \mapsto [[\overline{v}/\overline{x_j}]e_j]\mathcal{R}$, and evaluating the input to the recursive call yields $[\mathbf{let}\,\overline{x_j} = \overline{e}\,\mathbf{in}\,e_j]\mathcal{R} \mapsto^r$
$[\mathbf{let}\,\overline{x_j} = \overline{v}\,\mathbf{in}\,e_j]\mathcal{R} \mapsto [[\overline{v}/\overline{x_j}]e_j]\mathcal{R}$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 3.29 their ancestor terms are cost equivalent,
$[\mathbf{case}\,k_j\,\overline{e}\,\mathbf{of}\,\{p_i \rightarrow e_i\}]\mathcal{R} \trianglelefteq\!\trianglerighteq [\mathbf{let}\,\overline{x_j} = \overline{e}\,\mathbf{in}\,e_j]\mathcal{R}$, and cost equivalence implies strong improvement. By the induction hypothesis $[\mathbf{let}\,\overline{x_j} = \overline{e}\,\mathbf{in}\,e_j]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,\overline{x_j} = \overline{e}\,\mathbf{in}\,e_j]\!]\,\rho\,\mathcal{R})$, and therefore $[\mathbf{case}\,k_j\,\overline{e}\,\mathbf{of}\,\{p_i \rightarrow e_i\}]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![k_j]\!]\,\rho\,(\overline{\square\,e} : \mathbf{case}\ \square\ \ \mathbf{of}\,\{k_i\,\overline{x_i} \rightarrow e_i\} : \mathcal{R}))$.

B.4.5 R5

We have that $\rho(\mathcal{D}[\![\lambda\overline{x}.e_1]\!]\,\rho\,(\overline{\Box\ e_2} : \mathcal{R})) = \rho(\mathcal{D}[\![\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\!]\,\rho\,\mathcal{R})$. Evaluating the input term yields: $[(\lambda\overline{x}.e_1)\,\overline{e_2}]\mathcal{R} \mapsto^r [(\lambda\overline{x}.e_1)\,\overline{v}]\mathcal{R} \mapsto [[\overline{v}/\overline{x}]e_1]\mathcal{R}$, and evaluating the input to the recursive call yields: $[\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\mathcal{R} \mapsto^r [\mathbf{let}\,\overline{x} = \overline{v}\,\mathbf{in}\,e_1]\mathcal{R} \mapsto [[\overline{v}/\overline{x}]e_1]\mathcal{R}$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 3.29 their ancestor terms are cost equivalent, $[(\lambda\overline{x}.e_1)\,\overline{e_2}]\mathcal{R} \trianglelefteq\!\trianglerighteq [\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\mathcal{R}$, and cost equivalence implies strong improvement. By the induction hypothesis $[\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,\overline{x} = \overline{e_2}\,\mathbf{in}\,e_1]\!]\,\rho\,\mathcal{R})$, and therefore $[(\lambda\overline{x}.e_1)\,\overline{e_2}]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\lambda\overline{x}.e_1]\!]\,\rho\,(\overline{\Box\ e_2} : \mathcal{R}))$.

B.4.6 R6

We have that $\rho(\mathcal{D}[\![\lambda x.e]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{B}[\![\lambda x.e']\!]\,\rho\,\mathcal{R})$ where $e' = \mathcal{D}[\![e]\!]\,\rho\,\epsilon$. By the induction hypothesis $e \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]\,\rho\,\epsilon)$ and the conditions of the theorem ensure that $\mathit{fv}([\lambda x.e]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$, so $\rho(e) = e$. By the induction hypothesis, $[e']\mathcal{R}' \trianglerighteq_s \rho'(\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}')$ for all $e', \mathcal{R}', \rho'$ such that $|\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}'| < |\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}|$. This holds specifically for all $e', \mathcal{R}', \rho'$ such that $|\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}'| < |\mathcal{B}[\![\lambda x.e]\!]\,\rho\,\mathcal{R}|$, since $|\mathcal{B}[\![\lambda x.e]\!]\,\rho\,\mathcal{R}| < |\mathcal{D}[\![\lambda x.e]\!]\,\rho\,\mathcal{R}|$ by Lemma 3.18. The premise of Lemma 3.35 is now fulfilled, so we may conclude that $[\lambda x.e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![\lambda x.e]\!]\,\rho\,\mathcal{R})$. Taken together with congruence properties of improvement this allows us to conclude $[\lambda\overline{x}.e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\lambda x.e]\!]\,\rho\,\mathcal{R})$.

B.4.7 R7

We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x = n\,\mathbf{in}\,e]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{D}[\![[n/x]e]\!]\,\rho\,\mathcal{R})$. By the induction hypothesis $[[n/x]e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![[n/x]e]\!]\,\rho\,\mathcal{R})$, and since $[\mathbf{let}\,x = n\,\mathbf{in}\,e]\mathcal{R} \mapsto [[n/x]e]\mathcal{R}$ it follows from Lemma 3.31:4 that $[\mathbf{let}\,x = n\,\mathbf{in}\,e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,x = n\,\mathbf{in}\,e]\!]\,\rho\,\mathcal{R})$.

B.4.8 R8

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

B.4.9 R9

B.4.9.1 Case: $x \in \mathit{strict}(e_2)$ and $x \in \mathit{linear}(e_2)$

We have that $\rho(\mathcal{D}[\![\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{D}[\![[e_1/x]e_2]\!]\,\rho\,\mathcal{R})$. Evaluating the input term yields $[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R} \mapsto^r [\mathbf{let}\,x = v\,\mathbf{in}\,e_2]\mathcal{R} \mapsto [[v/x]e_2]\mathcal{R} \mapsto^s [v]\mathcal{E}$, and evaluating the input to the recursive call yields: $[[e_1/x]e_2]\mathcal{R} \mapsto^s [e_1]\mathcal{E} \mapsto^r [v]\mathcal{E}$. These two resulting terms are syntactically equivalent, and therefore cost equivalent. By Lemma 3.29 their ancestor terms are cost equivalent, $[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R} \trianglelefteq\!\trianglerighteq [[e_1/x]e_2]\mathcal{R}$, and cost equivalence implies strong improvement. By the induction hypothesis $[[e_1/x]e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![[e_1/x]e_2]\!]\,\rho\,\mathcal{R})$, and therefore $[\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\,x = e_1\,\mathbf{in}\,e_2]\!]\,\rho\,\mathcal{R})$.

B.4.9.2 Case: otherwise

We have that $\rho(\mathcal{D}[\![\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2]\!]\,\rho\,\mathcal{R}) = \rho(\mathbf{let}\, x = \mathcal{D}[\![e_1]\!]\,\rho\,\epsilon\,\mathbf{in}\,\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R})$, and the conditions of the proposition ensure that $\mathit{fv}([\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$, so $\rho(\mathbf{let}\, x = \mathcal{D}[\![e_1]\!]\,\rho\,\epsilon\,\mathbf{in}\,\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R}) = \mathbf{let}\, x = \rho(\mathcal{D}[\![e_1]\!]\,\rho\,\epsilon)\,\mathbf{in}\,\rho(\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R})$. By the induction hypothesis $e_1 \trianglerighteq_s \rho(\mathcal{D}[\![e_1]\!]\,\rho\,\epsilon)$ and $[e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e_2]\!]\,\rho\,\mathcal{R})$. By Lemma B.3 the input is strongly improved by $\mathbf{let}\, x = e_1 \,\mathbf{in}\, [e_2]\mathcal{R}$, and therefore $[\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2]\!]\,\rho\,\mathcal{R})$.

B.4.10 R10

We have that $\rho(\mathcal{D}[\![n]\!]\,\rho\,(\square \oplus e_2 : \mathcal{R})) = \rho(\mathcal{D}[\![e_2]\!]\,\rho\,(n \oplus \square : \mathcal{R}))$ and the conditions of the theorem ensure that $\mathit{fv}([n \oplus e_2]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so $\rho([n \oplus e_2]\mathcal{R}) = [n \oplus e_2]\mathcal{R}$ and by the induction hypothesis $[n \oplus e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e_2]\!]\,\rho\,(n \oplus \square : \mathcal{R}))$.

B.4.11 R11

We have that $\rho(\mathcal{D}[\![e_1 \oplus e_2]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{D}[\![e_1]\!]\,\rho\,(\square \oplus e_2 : \mathcal{R}))$ and the conditions of the theorem ensure that $\mathit{fv}([e_1 \oplus e_2]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so $\rho([e_1 \oplus e_2]\mathcal{R}) = [e_1 \oplus e_2]\mathcal{R}$ and by the induction hypothesis $[e_1 \oplus e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e_1]\!]\,\rho\,(\square \oplus e_2 : \mathcal{R}))$.

B.4.12 R12

We have that $\rho(\mathcal{D}[\![e_1\, e_2]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{D}[\![e_1]\!]\,\rho\,(\square\; e_2 : \mathcal{R}))$ and the conditions of the theorem ensure that $\mathit{fv}([e_1\, e_2]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so $\rho([e_1\, e_2]\mathcal{R}) = [e_1\, e_2]\mathcal{R}$ and by the induction hypothesis $[e_1\, e_2]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e_1]\!]\,\rho\,(\square\; e_2 : \mathcal{R}))$.

B.4.13 R13

We have that $\rho(\mathcal{D}[\![\mathbf{case}\, e\, \mathbf{of}\, \{p_i \rightarrow e_i\}]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{D}[\![e]\!]\,\rho\,(\mathbf{case}\;\square\;\mathbf{of}\,\{p_i \rightarrow e_i\} : \mathcal{R}))$ and the conditions of the theorem ensure that $\mathit{fv}([\mathbf{case}\, e\, \mathbf{of}\, \{p_i \rightarrow e_i\}]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so
$\rho([\mathbf{case}\, e\, \mathbf{of}\, \{p_i \rightarrow e_i\}]\mathcal{R}) = [\mathbf{case}\, e\, \mathbf{of}\, \{p_i \rightarrow e_i\}]\mathcal{R}$ and by the induction hypothesis
$[\mathbf{case}\, e\, \mathbf{of}\, \{p_i \rightarrow e_i\}]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]\,\rho\,(\mathbf{case}\;\square\;\mathbf{of}\,\{p_i \rightarrow e_i\} : \mathcal{R}))$.

B.4.14 R14

We have that $\rho(\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}) = \rho(\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R})$ and the conditions of the theorem ensure that $\mathit{fv}([e]\mathcal{R}) \cap \mathit{dom}(\rho) = \emptyset$ so $\rho([e]\mathcal{R}) = [e]\mathcal{R}$. By the induction hypothesis, $[e']\mathcal{R}' \trianglerighteq_s \rho'(\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}')$ for all $e', \mathcal{R}', \rho'$ such that $|\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}'| < |\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}|$. This holds specifically for all $e', \mathcal{R}', \rho'$ such that $|\mathcal{D}[\![e']\!]\,\rho'\,\mathcal{R}'| < |\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}|$, since $|\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R}| < |\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R}|$ by Lemma 3.18. The premise of Lemma 3.35 is now fulfilled, so we may conclude that $[e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{B}[\![e]\!]\,\rho\,\mathcal{R})$. Taken together with congruence properties of improvement this allows us to conclude $[e]\mathcal{R} \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]\,\rho\,\mathcal{R})$.

References

M. Abadi and B. Lampson J-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 83–91, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7.

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

E. Albert, J. P. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based homeomorphic embedding for online termination. *Inf. Process. Lett*, 109(15):879–886, 2009.

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

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

A. M. Ben-Amram and C. S. Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst*, 29(1), 2007.

M. Bolingbroke. Re: ANN: HLint 1.2. URL `http://markmail.org/message/bdc3pjhneuatrd26`. Email to the Haskell-Cafe mailing list, January 2009.

M. Bolingbroke and S. L. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 135–146. ACM, 2010.

E. C. Brady and K. Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 297–308, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

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.

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

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

K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.

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

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.

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

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 A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. *Lecture Notes in Computer Science*, 724:112–123, 1993. ISSN 0302-9743.

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. 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.

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

G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

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.

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

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

P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher-order call-by-value language. *Logical Methods in Computer Science*, 6(3), 2010a.

P. A. Jonsson and J. Nordlander. Strengthening Supercompilation for Call-By-Value Languages. In *Second International Workshop on Metacomputation in Russia*, pages 64–81, 2010b.

P. A. Jonsson and J. Nordlander. Taming code explosion in supercompilation. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 33–42. ACM, 2011. ISBN 978-1-4503-0485-6.

I. Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics, Moscow, 2010.

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

I. Klyuchnikov and S. Romanenko. Towards Higher-Level Supercompilation. In *Second International Workshop on Metacomputation in Russia*, pages 82–101, 2010.

I. Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions, 2011. (Submitted).

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.

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

C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'01*, pages 81–92, 2001.

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

M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In T. Æ. Mogensen, D. A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 379–403. Springer, 2002. ISBN 3-540-00326-6.

M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'08)*, pages 119–134. Springer LNCS 5438, 2009.

M. Leuschel, D. Elphick, M. Varea, S-J Craig, and M. Fontaine. The ecce and logen partial evaluators and their web interfaces. In *PEPM*, pages 88–94, New York, NY, USA, 2006. ACM. ISBN 1-59593-196-1.

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

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.

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

N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, June 2008.

N. Mitchell. Rethinking supercompilation. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 309–320. ACM, September 2010. ISBN 978-1-60558-794-3.

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.

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

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

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 *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

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

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.

S. L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program*, 12(4&5):393–433, 2002.

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

S. L. Peyton Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 165–176, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7.

F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

J. S Reich, M. Naylor, and C. Runciman. Supercompilation and the Reduceron. In *Second International Workshop on Metacomputation in Russia*, pages 159–172, 2010.

S. A. Romanenko. Arity Raiser and Its Use in Program Specialization. In *ESOP'90, Copenhagen, Denmark*, volume 432 of *LNCS*, pages 341–360. Springer-Verlag, 1990.

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.

A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995.

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

D. Sereni. Termination analysis and call graph construction for higher-order functional programs. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 71–84. ACM, 2007. ISBN 978-1-59593-815-2.

P. Sestoft. Compiling spreadsheet-defined functions, 2010a. URL `http://www.itu.dk/~sestoft/papers/sestoft-spreadsheet.pdf`. Draft.

P. Sestoft. Partial evaluation of spreadsheet-defined functions, 2010b. URL `http://meta2010.pereslavl.ru/program/meta2010-Peter-Sestoft.pdf`. Presentation at META'10.

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*, pages 465–479. 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.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international worksh op on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X.

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

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

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.

G. Vidal. A Hybrid Approach to Conjunctive Partial Deduction. In *Proc. of the 21th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'10)*, 2010. Available from `http://users.dsic.upv.es/~gvidal/german/papers.html`.

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.