Supercompiling Overloaded Functions

Peter A. Jonsson Johan Nordlander
Luleå University of Technology
{pj, nordland}@csee.ltu.se

Abstract

A naïve translation of recursive cases in type class instances will give code that allocates a new dictionary and copies the data from the input dictionary to the new dictionary for the recursive call. This record creation allocates memory and leads to bad performance for a conceptually simple and rather common operation, for example equality between lists. We present a positive supercompilation algorithm, parametrized with respect to evaluation order, and show how this algorithm can remove these allocations resulting in both time and space savings. We also show how a small extension to the folding mechanism of our supercompiler can give similar effects the static argument transformation at nearly no extra cost.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Compilers, Optimization; D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages

General Terms Languages, Theory

Keywords supercompilation, type classes, static argument transformation

1. Introduction

Type Classes (Wadler and Blott 1989) is a mechanism to support a combination of overloading and polymorphism. A program with type classes can be translated at compile-time to an equivalent program without type classes. The translated code adds a hidden parameter called a dictionary to overloaded functions, sometimes in the form of a record. This dictionary contains references to the functions in that particular type class instance.

Translating recursive type classes introduces additional problems: Wadler and Blott notes that "translation of the [Eq] instance declaration over lists is a little trickier", where the instance declaration is:

$$
\begin{array}{lll}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} & & \\
[] & == [] & = True \\
x : xs & == y : ys & = x == y \mathbin{\&\&} xs == ys \\
\_ & == \_ & = False
\end{array}
$$

A naïve translation of the recursive case will give code that allocates a new dictionary and copies the data from the input dictionary to the new dictionary for the recursive call. This will lead to bad performance for a conceptually simple and rather common operation.

Wadler and Blott suggests that "it is easy for the compiler to perform beta reductions to transform terms [on a certain form]", but does not provide a transformation. Augustsson (1993) describes several straightforward techniques that Haskell compilers implemented at the time, and if these optimizations were combined they could have a significant impact on performance. Two key components of his work is inlining (Peyton Jones and Marlow 2002) and function specialization (Peyton Jones 2007), two difficult problems that have been extensively studied.

Around the same time Jones (1995) used partial evaluation to remove the run-time passing of dictionaries by generating specialized versions of overloaded functions, and somewhat surprisingly this led to a reduction of size of the compiled programs. While his work discusses polymorphic recursion it does not provide an explicit solution to the problem since neither Haskell nor Gofer allowed such programs at the time.

In this paper, we show how *positive supercompilation* (Jonsson and Nordlander 2009), a program transformation that performs program specialization and removes intermediate structures, can remove the dictionary allocations in the instances for recursive data types. Our previous work was specific to call-by-value, but we extend this work to be parametric over the evaluation order. The examples we show are ordinary instances found in real code. Since polymorphically recursive functions must have their dictionaries available at run-time we can not remove these dictionaries, but the supercompiler is guaranteed to terminate in the presence of polymorphic recursion.

We also show how a small adjustment to the folding mechanism of our supercompiler allows us to achieve similar effects to the static argument transformation (SAT) (Santos 1995). The added compilation cost of our modification compared to ordinary supercompilation is small: it requires comparing two lists of free variables at function call sites and generating a few more functions. Whether this is beneficial depends on the compiler backend: a backend with mandatory lambda lifting (Johnsson 1985) will not benefit, but it appears GHC can take advantage of this on the nofib benchmark suite (Partain 1992) where it gives a 12% speedup (Bolingbroke 2009). Under call-by-need with the right backend we believe our extension gives a good power to weight ratio.

The specific contributions of our work are:

- We simplify our previous work on positive supercompilation (Jonsson and Nordlander 2009), parametrize the algorithm with respect to evaluation strategy, and extend it to a language that includes records in order to represent dictionaries. We also prove this extension correct. (Section 4).
- We explain the problem of translating recursive type classes and quantify the performance difference between naïve translations and supercompiled naïve translations in a strict language (Section 5).

- We extend the folding mechanism of the positive supercompiler to also include SAT and prove that it terminates (Section 6).

We give a brief survey of positive supercompilation in Section 2. Our language of study is defined in Section 3, right before the technical contributions are presented.

2. Survey of Positive Supercompilation

A supercompiler is an interpreter that can handle open terms without getting stuck, combined with some detection of when the transformation might be diverging. Our first example shows how applications are handled and how the folding mechanism works in the simple recursive case. The supercompiler will not be able to optimize this expression, but we get back to this example in Section 6.1 since it is one of the simplest expressions that demonstrate how our modified supercompiler works. We start with the standard definition of map:

$$\begin{array}{l} map = \lambda f\ xs.\ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (x : xs) \rightarrow f\ x : map\ f\ xs \end{array}$$

To transform the expression $map\ f'\ ys$ we start by allocating a new fresh function name (h_0) and inline the body of map:

$$\begin{array}{l} (\lambda f\ xs.\ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (x : xs) \rightarrow f\ x : map\ f\ xs)\ f'\ ys \end{array}$$

which we rewrite to a let-statement with the binders from the lambda expression bound to the arguments of the application:

$$\begin{array}{l} \mathbf{let}\ xs = ys,\ f = f'\ \mathbf{in}\ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad\qquad\qquad [] \rightarrow [] \\ \qquad\qquad\qquad\qquad (x : xs) \rightarrow f\ x : map\ f\ xs \end{array}$$

If this let-statement was binding general expressions it would be unsafe to blindly substitute into the body under call-by-value since it might remove non-termination in the original program, but still safe under call-by-name: this highlights the key difference between positive supercompilation for call-by-value and call-by-name. Both *ys* and *f'* are variables though, so for this example it is perfectly safe to substitute regardless of evaluation order:

$$\begin{array}{ll} \mathbf{case}\ ys\ \mathbf{of} & (*) \\ \quad [] \rightarrow [] & \\ \quad (x : xs) \rightarrow f'\ x : map\ f'\ xs & \end{array}$$

We will return to this equation in Section 6.1, and will refer to it as $(*)$. We can not improve the head of the case-statement, so we proceed to transform the branches: the first branch does not contain any interesting expressions, but the second branch contains a recursive function call that looks quite similar to the expression we started out with. In fact, that expression is a renaming of what we started out with, and we therefore fold: replace the call $map\ f'\ xs$ with $h_0\ f'\ xs$. The complete result of the transformation is:

$$\begin{array}{l} \mathbf{letrec}\ h_0 = \lambda f'\ xs.\ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad\qquad\qquad [] \rightarrow [] \\ \qquad\qquad\qquad\qquad (x : xs) \rightarrow f'\ x : h_0\ f'\ xs \\ \mathbf{in}\ h_0\ f'\ xs \end{array}$$

As we can see this is just a new function h_0 that is isomorphic to the original map, and a new call to that function instead. Nevertheless, we have seen most of the important building blocks of positive supercompilation already.

Our second example will remove an intermediate list, by only introducing one new building block of positive supercompilation: the case-of-case transformation. We will skip writing out some intermediate steps of the transformation since they are similar to the previous example. We need the standard definition of zip:

$$\begin{array}{l} zip = \lambda xs\ ys.\ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad [] \rightarrow [] \\ \qquad (x' : xs') \rightarrow \mathbf{case}\ ys\ \mathbf{of} \\ \qquad\qquad\qquad [] \rightarrow [] \\ \qquad\qquad\qquad (y' : ys') \rightarrow \\ \qquad\qquad\qquad\qquad (x', y') : zip\ xs'\ ys' \end{array}$$

To transform the expression $zip\ (map\ f'\ zs)\ zs'$ we start by allocating a new fresh function name (h_1). We then inline the body of zip, rewrite to a let-statement just like the previous example:

$$\begin{array}{l} \mathbf{let}\ xs = map\ f'\ zs,\ ys = zs' \\ \mathbf{in}\ \mathbf{case}\ xs\ \mathbf{of} \\ \quad [] \rightarrow [] \\ \quad (x' : xs') \rightarrow \mathbf{case}\ ys\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys' \end{array}$$

For this step it is safe to substitute $map\ f'\ zs$ for *xs* regardless of evaluation order since the body of the let-statement is strict with respect to *xs*. After substituting the expression is:

$$\begin{array}{l} \mathbf{case}\ map\ f'\ zs\ \mathbf{of} \\ [] \rightarrow [] \\ (x' : xs') \rightarrow \mathbf{case}\ zs'\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys' \end{array}$$

We focus on the head of the outermost case, and inline map, rewrite it to a let-statement and perform the substitution, to end up with:

$$\begin{array}{l} \mathbf{case}\ (\ \mathbf{case}\ zs\ \mathbf{of} \\ \qquad [] \rightarrow [] \\ \qquad (x : xs) \rightarrow f'\ x : map\ f'\ xs)\ \mathbf{of} \\ [] \rightarrow [] \\ (x' : xs') \rightarrow \mathbf{case}\ zs'\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys' \end{array}$$

It is time to introduce the case-of-case rule: we push the outer case-statement into each branch of the inner case-statement:

$$\begin{array}{l} \mathbf{case}\ zs\ \mathbf{of} \\ [] \rightarrow \mathbf{case}\ []\ \mathbf{of} \\ \qquad [] \rightarrow [] \\ \qquad (x' : xs') \rightarrow \mathbf{case}\ zs'\ \mathbf{of} \\ \qquad\qquad\qquad [] \rightarrow [] \\ \qquad\qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys' \\ (x : xs) \rightarrow \mathbf{case}\ f'\ x : map\ f'\ xs\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (x' : xs') \rightarrow \mathbf{case}\ zs'\ \mathbf{of} \\ \qquad\qquad\qquad\qquad [] \rightarrow [] \\ \qquad\qquad\qquad\qquad (y' : ys') \rightarrow \\ \qquad\qquad\qquad\qquad\qquad (x', y') : zip\ xs'\ ys' \end{array}$$

We can see that the first branch of the outer case-statement reduces to the empty list, and the second branch reduces to a case-statement, taking both steps at once gives:

$$\begin{array}{l} \mathbf{case}\ zs\ \mathbf{of} \\ [] \rightarrow [] \\ (x : xs) \rightarrow \mathbf{case}\ zs'\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (y' : ys') \rightarrow (f'\ x, y') : zip\ (map\ f'\ xs)\ ys' \end{array}$$

Once again we encounter a renaming of what we started out with in the second branch, so we replace $zip\ (map\ f'\ xs)\ ys$ with $h_1\ f'\ xs\ ys$ and create a new function, with the end result:

$$\mathbf{letrec}\ h_1 = \lambda\ zs\ zs'.$$

Expressions

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

Values

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

Figure 1. The language

$$
\begin{array}{l}
\textbf{case}\ zs\ \textbf{of} \\
\quad [] \to [] \\
\quad (x : xs) \to \textbf{case}\ zs'\ \textbf{of} \\
\qquad\qquad [] \to [] \\
\qquad\qquad (y' : ys') \to (f'\ x, y') : h_1\ f'\ xs\ ys' \\
\textbf{in}\ h_1\ f'\ zs\ zs'
\end{array}
$$

Should these two examples not be sufficient there are more examples of written out transformations in the work by Wadler (1990), Sørensen et al. (1996), and Jonsson and Nordlander (2009)

3. Language

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

Let X be an enumerable set of variables ranged over by x, N the set of integers ranged over by n, and K a set of constructor symbols ranged over by k. Let g range over an enumerable set of defined names and let $\mathcal{G}$ be a given set of recursive definitions of the form $(\overline{g}, \overline{e})$. The language contains arithmetic operations $\oplus$ on the integers, although these meta-variables can preferably be understood as ranging over primitive values in general and arbitrary operations on these. We let $+$ denote the semantic meaning of $\oplus$. We construct records by $\{\}$ and record selection by . assuming an enumerable set of label identifiers ranged over by l.

We abbreviate a list of expressions $e_1 \ldots e_n$ as $\overline{e}$, and a list of variables $x_1 \ldots x_n$ as $\overline{x}$. All functions have a specific arity and all applications must be saturated; hence $\lambda x.map\ (\lambda y.y\ +\ 1)\ x$ is legal whereas $map\ (\lambda y.y\ +\ 1)$ is not. We denote the free variables of an expression e by $fv(e)$, and function names $fn(e)$.

A program is an expression with no free variables and all function names defined in $\mathcal{G}$. We denote capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e' by $[\overline{e}/\overline{x}]e'$.

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

4. Positive Supercompilation

This section contains a simplified and more declarative version of our previous work (Jonsson and Nordlander 2009), parametrized with respect to evaluation order. This revision includes a correction to a case in rule R16 where our previous call-by-value supercompiler could introduce termination into programs despite our efforts to avoid it. We extend our work with records in order to represent the dictionaries necessary for type classes, and prove the extension correct.

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

Figure 3. The strict variables of an expression

Our supercompiler is defined as a set of rewrite rules that pattern-match on expressions. This algorithm is called the *driving* algorithm, and is defined in Figure 2. Two additional parameters appear as subscripts to the rewrite rules: a memoization list ρ and a driving context $\mathcal{R}$. The memoization list holds information about expressions already traversed and is explained more in detail in Section 4.1. The driving context $\mathcal{R}$ is defined as follows:

$$
\mathcal{R} ::= [\,] \mid \mathcal{R}\,\overline{e} \mid \textbf{case}\, \mathcal{R}\, \textbf{of}\, \{p_i \to e_i\} \mid \mathcal{R} \oplus e \mid e \oplus \mathcal{R} \mid \mathcal{R}.l
$$

A driving context $\mathcal{R}$ is a term containing a single hole $[\,]$, which indicates the next expression to be transformed. The expression $\mathcal{R}\langle e \rangle$ is the term obtained by replacing the hole in $\mathcal{R}$ with e.

An expression e is strict with regards to a variable x if it eventually evaluates x; in other words, if $e \mapsto \ldots \mapsto \mathcal{E}\langle x \rangle$. Such information is not computable in general, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 3, where the strict variables of an expression e are defined as all free variables of e except those that only appear under a lambda or not inside all branches of a case.

There are three rules that differ between call-by-value and call-by-name: R16, R17 and R18. It is safe to substitute regardless of strictness properties in both rule R16 and rule R17 under call-by-name. We assume that the right hand side of letrec bindings in rule R18 are all values under call-by-value, but allow arbitrary expressions under call-by-name. This restriction can be lifted under call-by-value (Nordlander et al. 2008b) as well, but our supercompiler needs further extensions to handle this.

There is an ordering between rules; i.e., all rules must be tried in the order they appear. Rules R14 and R23 are the default fallback cases which extend the given driving context $\mathcal{R}$ and zoom in on the next expression to be driven. The program is turned "inside-out" by moving the surrounding context $\mathcal{R}$ into all branches of the case-statement through rules R19 and R22. Rule R17 has a similar mechanism for let-statements. Notice how the context is moved out of the recursive call in rule R5, whereas rule R8 recursively applies the driving algorithm to the full new term $\mathcal{R}\langle n \rangle$, forcing a re-traversal of the new term in hope of further reductions. Meta-variable a in rule R9, R11 and rule R22 stands for an "annoying" expression; i.e., an expression that would be further reducible were it not for a free variable getting in the way. The grammar for annoying expressions is:

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

Some expressions should be handled differently depending on context. If a constructor application appears in an empty context, there is not much we can do but to drive the argument expressions (rule R4). On the other hand - if the application occurs at the head of a case expression, we may choose a branch on the basis of the

$$
\begin{array}{lcll}
\mathcal{D}[\![n]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle n\rangle & \text{(R1)}\\
\mathcal{D}[\![x]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle x\rangle & \text{(R2)}\\
\mathcal{D}[\![g]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}_{app}(g,)_{\mathcal{R},\mathcal{G},\rho} & \text{(R3)}\\
\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{G},\rho} & = & k\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho} & \text{(R4)}\\
\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\mathcal{G},\rho}\rangle & \text{(R5)}\\
\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{G},\rho} & = & (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}) & \text{(R6)}\\
\mathcal{D}[\![\{l_i = e_i\}]\!]_{[],\mathcal{G},\rho} & = & \{l_i = \mathcal{D}[\![e_i]\!]_{[],\mathcal{G},\rho}\} & \text{(R7)}\\
\mathcal{D}[\![n_1 \oplus n_2]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle n\rangle]\!]_{[],\mathcal{G},\rho}, \text{ where } n = n_1 + n_2 & \text{(R8)}\\
\mathcal{D}[\![e_1 \oplus e_2]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e_1]\!]_{[],\mathcal{G},\rho} \oplus \mathcal{D}[\![e_2]\!]_{[],\mathcal{G},\rho}, \text{ if } e_1 \oplus e_2 = a & \text{(R9)}\\
 & & \mathcal{D}[\![e_2]\!]_{\mathcal{R}\langle e_1 \oplus []\rangle,\mathcal{G},\rho}, \text{ if } e_1 = n \text{ or } e_1 = a & \\
 & & \mathcal{D}[\![e_1]\!]_{\mathcal{R}\langle []\oplus e_2\rangle,\mathcal{G},\rho}, \text{ otherwise} & \\
\mathcal{D}[\![\{l_i = e_i\}.l_j]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let } x_i \backslash x_j = e_i \backslash e_j \textbf{ in } e_j]\!]_{\mathcal{R},\mathcal{G},\rho},\ x_i \text{ fresh} & \text{(R10)}\\
\mathcal{D}[\![e.l_j]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle\mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho}.l_j\rangle, \text{ if } e = a & \text{(R11)}\\
 & & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle[].l_j\rangle,\mathcal{G},\rho}, \text{ otherwise} & \\
\mathcal{D}[\![g\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & \text{(R12)}\\
\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let } \overline{x} = \overline{e} \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & \text{(R13)}\\
\mathcal{D}[\![e\,\overline{e}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,\overline{e}\rangle,\mathcal{G},\rho} & \text{(R14)}\\
\mathcal{D}[\![\textbf{let } x = v \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R15)}\\
\mathcal{D}[\![\textbf{let } x = y \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\mathcal{G},\rho}, \text{ if } (y \notin \mathit{SPLIT}) \text{ or call-by-name} & \text{(R16)}\\
 & & \textbf{let } x = y \textbf{ in } \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}, \text{ otherwise} & \\
\mathcal{D}[\![\textbf{let } x = e \textbf{ in } f]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle[e/x]f\rangle]\!]_{[],\mathcal{G},\rho}, \text{ if } (x \in \mathit{strict}(f) \text{ and f linear w.r.t } x) \text{ or call-by-name} & \text{(R17)}\\
 & & \textbf{let } x = \mathcal{D}[\![e]\!]_{[],\mathcal{G},\rho} \textbf{ in } \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{G},\rho}, \text{ otherwise} & \\
\mathcal{D}[\![\textbf{letrec } g = e \textbf{ in } e]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{letrec } g = e \textbf{ in } e', \text{ if } g \in \mathit{fn}(e') & \text{(R18)}\\
 & & e', \text{ otherwise} & \\
 & & \text{where } e' = \mathcal{D}[\![\mathcal{R}\langle e\rangle]\!]_{[],\mathcal{G}',\rho} \text{ and } \mathcal{G}' = \mathcal{G} \cup (g,e) & \\
\mathcal{D}[\![\textbf{case } x \textbf{ of } \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{case } x \textbf{ of } \{p_i \to \mathcal{D}[\![\mathcal{R}\langle[p_i/x]e_i\rangle]\!]_{[],\mathcal{G},\rho}\} & \text{(R19)}\\
\mathcal{D}[\![\textbf{case } k_j\,\overline{e} \textbf{ of } \{k_i\,\overline{x}_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\textbf{let } \overline{x}_j = \overline{e} \textbf{ in } e_j]\!]_{\mathcal{R},\mathcal{G},\rho} & \text{(R20)}\\
\mathcal{D}[\![\textbf{case } n_j \textbf{ of } \{n_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![\mathcal{R}\langle e_j\rangle]\!]_{[],\mathcal{G},\rho} & \text{(R21)}\\
\mathcal{D}[\![\textbf{case } a \textbf{ of } \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{case } \mathcal{D}[\![a]\!]_{[],\mathcal{G},\rho} \textbf{ of } \{p_i \to \mathcal{D}[\![\mathcal{R}\langle e_i\rangle]\!]_{[],\mathcal{G},\rho}\} & \text{(R22)}\\
\mathcal{D}[\![\textbf{case } e \textbf{ of } \{p_i \to e_i\}]\!]_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{D}[\![e]\!]_{\mathcal{R}\langle\textbf{case } [] \textbf{ of } \{p_i \to e_i\}\rangle,\mathcal{G},\rho} & \text{(R23)}
\end{array}
$$

Figure 2. Driving algorithm

constructor and leave the arguments unevaluated in the hope of finding fold opportunities further down the syntax tree (rule R20).

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

Notice that the primitive operations ranged over by $\oplus$ can not be unfolded and transformed like ordinary functions can. If the arguments of a primitive operation are annoying our algorithm will simply leave the primitive operation in place (rule R9).

When a known selector is applied to a record we create a let-statement with fresh binders and the remaining expressions in the record bound to them, and the selected expression as body (rule R10). If those expressions are values they will be transformed away by rule R15, otherwise they must remain to preserve termination properties under call-by-value. It is safe to remove them under call-by-name. There is a fallback rule for extending the context in case a selector is applied to an expression that is not a record (rule R11). Upon encountering a record without any selectors nearby our supercompiler simply leaves the context and transforms each member expression of the record (rule R7).

4.1 Application Rule

In the driving algorithm rule R3 and rule R12 refer to $\mathcal{D}_{app}(\,)$, defined in Figure 4. $\mathcal{D}_{app}(\,)$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation. Figure 4 contains some new notation: we use σ for a variable to variable substitution and $=$ for syntactical equivalence of expressions.

Care needs to be taken to ensure that recursive functions are not inlined forever. The driving algorithm keeps a record of previously encountered applications in the memoization list ρ; whenever it detects an expression that is equivalent (up to renaming of variables) to a previous expression, the algorithm creates a new recursive function h_n for some n. Whenever such an expression is encountered again, a call to h_n is inserted.

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

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

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

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

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

$$\frac{e_1 \trianglelefteq f_1, \ldots, e_n \trianglelefteq f_n}{s(e_1, \ldots, e_n) \trianglelefteq s(f_1, \ldots, f_n)}$$

Examples of the homeomorphic embedding are shown in Figure 5.

$$\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} = h\,\overline{x} \quad \text{if } \exists(h,t)\in\rho\,.\,\sigma t = \mathcal{R}\langle g\,\overline{e}\rangle \quad (1)$$
where $\overline{x} = fv(\mathcal{R}\langle g\,\overline{e}\rangle)$

$$\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} = \mathcal{R}\langle g\,\overline{e}\rangle \quad \text{if } \exists(h,t)\in\rho\,.\,t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \text{ and } \mathcal{R}\langle g\,\overline{e}\rangle \trianglelefteq t \quad (2)$$

$$\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} = [\mathcal{D}[\![\overline{t}]\!]_{[],\mathcal{G},\rho}/\overline{y}]\mathcal{D}[\![t_g]\!]_{[],\mathcal{G},\rho} \quad \text{if } \exists(h,t)\in\rho\,.\,t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle \quad (3)$$
where $(t_g,\overline{t},\overline{y}) = split(\mathcal{R}\langle g\,\overline{e}\rangle, t)$ and $\mathcal{C}' = \mathcal{C}\cup\overline{y}$

$$\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} = \textbf{letrec}\ h = \lambda\overline{x}.e'\ \textbf{in}\ h\,\overline{x} \quad \text{if } h\in fn(e') \quad (4a)$$
$$e' \quad \text{otherwise} \quad (4b)$$
where $(g,e)\in\mathcal{G}$, $e' = \mathcal{D}[\![\mathcal{R}\langle e\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'}$, $\overline{x} = fv(\mathcal{R}\langle g\,\overline{e}\rangle)$, $\rho' = \rho\cup(h,\mathcal{R}\langle g\,\overline{e}\rangle)$ and h fresh

Figure 4. Driving of applications

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

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

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

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

We refer to t_g as the ground term. For background information and an algorithm to compute most specific generalizations, see Lassez et al. (1988). Figure 5 also contains examples of the msg.

The most specific generalization is not always sufficient to split expressions. For some expressions it will return the ground term as a variable, and the respective θs equal to the input terms. If this happens we need to split expressions in a different way. We define a function split that has a guard against the ground term being a variable, and returns the spine of the term and its subterms. It is important that split creates fresh variables from the set *SPLIT* because these variables will be the target of substitutions which cannot be allowed to change termination properties:

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

$$split(s(\overline{e}_1),s'(\overline{e}_2)) = (s(\overline{x}),\overline{e}_1,\overline{x}) \quad \textit{if } s\neq s'$$
$$= (t_g,\overline{e},\overline{y}) \quad \textit{otherwise}$$

with $(t_g,[\overline{e}/\overline{y}],\_) = msg(s(\overline{e}_1),s'(\overline{e}_2))$ *and* $\overline{x}$, $\overline{y}$ *fresh from* SPLIT.

All the examples of how our algorithm works in Section 2 eventually terminate through a combination of alternative 1 and alternative 4a of $\mathcal{D}_{app}(\,)$.

Alternative 3 is for downwards generalization, when terms are "growing" in some sense we make sure to split them. An example of *reverse* with an accumulating parameter is shown in Figure 6 with the definition of reverse as:

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

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

(Put $(h_0,\ rev\ xs\ [])$ in ρ and transform the program according to the rules of the algorithm)

$$\textbf{case}\ xs\ \textbf{of}\ [] \to []\ ;\ (x':xs') \to \mathcal{D}[\![\ rev\ xs'\ (x':[])]\!]$$

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

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

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

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

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

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

(Putting the two parts together)

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

Figure 6. Example of downwards generalization

4.2 Correctness

The algorithm both terminates and preserves the semantics. The theorems from our previous work can still be used, but the proofs have to be updated. The general structure of the proof is to prove that the algorithm is total, and that each step must decrease the weight of the term being transformed. The weight can be thought of as the size of the term being transformed.

The same theorems can be used for the call-by-name algorithm as well, but the proofs need to be updated. The notion of weight needs to be adjusted for the call-by-name case to account for the number of times the variable bound in a let-statement is used in its body. The proofs of correctness for call-by-name are similar in structure to those by Sands (1996).

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

Figure 5. Examples of the homeomorphic embedding and the msg

Lemma 4.5 (Totality)**.** *For all well-typed expressions e, $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}$ is matched by a unique rule in Figure 2.*

Proposition 4.6 (Termination)**.** *The driving algorithm $\mathcal{D}[\![\ ]\!]$ terminates for all well-typed inputs.*

To prove that the algorithm does not alter the semantics we use the improvement theory (Sands 1997). We define the standard notions of operational approximation and equivalence. A general context C which has zero or more holes in the place of some subexpressions is introduced.

Definition 4.7 (Operational Approximation and Equivalence)**.**

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

Notice that improvement $\trianglerighteq$ below is not the same as the homeomorphic embedding $\trianglelefteq$ defined previously. We use Sands's definitions for improvement and strong improvement:

Definition 4.8 (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 $e \cong e'$.*

which allows us to state the final theorem:

Proposition 4.9 (Total Correctness)**.** *Let e be an expression, and ρ an environment such that*

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

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

4.3 Extensions

Readers already familiar with supercompilation will have noticed that the upwards generalization is missing from this algorithm. This ability can be recovered by adding a new alternative to $\mathcal{D}_{app}()$ before the current alternative 4a, leaving the rest of $\mathcal{D}_{app}(\ )$ intact:

$$\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} = [\mathcal{D}[\![\overline{t}]\!]_{[],\mathcal{G},\rho}/\overline{y}]\mathcal{D}[\![t_g]\!]_{[],\mathcal{G},\rho} \text{ (4a)} \quad \text{if } \exists t \in e'.t \trianglelefteq \mathcal{R}\langle g\,\overline{e}\rangle$$
$$\text{where } (t_g, \overline{t}, \overline{y}) = split(\mathcal{R}\langle g\,\overline{e}\rangle, t)$$

This is essentially backtracking: when alternative 2 of $\mathcal{D}_{app}()$ triggers, the new 4a is there to catch this case and start over with a better expression. However, this is not a central issue for the work presented here, and we quickly move on to supercompiling overloaded functions.

5. Translating Recursive Type Classes

The straightforward translation of the Eq instance for lists, similar to the one given by Wadler and Blott (1989), is shown in Figure 7. We use Haskell syntax for these examples as convenience for the reader. The variable *w* is the dictionary, brackets {} are for record construction and the field name applied to a record is for record selection. An interesting observation is the piece of code $(eq\ (eqList\ w))$ which will construct a new record of $eqList\ w$ to immediately tear it down by record selection of the *eq* member. This both takes time and creates unnecessary work for the garbage collector.

We now substantiate our claim that a supercompiler actually can remove the dictionary allocations in the Eq-instance for lists. A supercompiled right hand side of eqList is quite different, as shown in Figure 8. We have left out the code for *neq* for space reasons, but the same effects occur there: dictionary creation and destruction no longer occur in every recursive call.

Some spurious case-statements appear first, and then a call to a new function h_1. This new function takes the dictionary as parameter, but the construction and destruction of the dictionary for every recursive call has vanished. This is still a function that is polymorphic: the same function can be used for comparing both lists of characters and lists of integers. Comparison of similar lists longer than one entry will spend most of the execution time in h_1. There are two let-statements in h_1 that are necessary for preserving termination properties under call-by-value with recursive lists (Nordlander et al. 2008b), but the outcome of the test in case it terminates is known in advance.

The program we supercompiled creates two lists of equal length and contents, compares them, and prints the result. A preliminary hint about performance of our supercompiler: compiling this program on an idle 2 Ghz Intel Core Duo processor takes 0.45 seconds according to the "time" command and turning on the supercompiler for the eqList instance increases the total compilation time to 0.47 seconds. In other words: our current untuned supercompiler implementation with some debugging code left in takes 0.02 seconds to supercompile eqList.

Our next example is a pretty printer from the Programatica project (Hallgren 2003) shown in Figure 9. The goal was to define an extensible abstract syntax for expressions and values by separating the constructors in an abstract data type from the operation that creates a recursive type. Programatica chose to define non-recursive data types with constructors for each form, and abstract over an extra type parameter that is used to tie the knot of the recursive type. We have removed everything except applications and identifiers in this example.

The call-by-name algorithm is stronger than the call-by-value algorithm which shows in this example: the call-by-name algorithm removes the dictionary allocations as shown in Figure 10. For space reasons of our type setting we have put the new function h_2 at the top level while it really should be local in pExp. We believe this limitation for call-by-value stems from our current formulation of the algorithm and are working on a reformulation that would allow a call-by-value algorithm to give a similar result as the call-by-name algorithm for this example.

One could make a compiler pragma available for supercompiling certain definitions, similar to pragmas available today for forcing the compiler to inline certain functions. This would give the programmer both the possibility and responsibility to control how large parts of the program that should be supercompiled. Functions that are too expensive to supercompile would simply not be annotated with this pragma.

```
data MyEq a = MyEq { eq :: a -> a -> Bool, neq :: a -> a -> Bool }

eqList      :: MyEq a -> MyEq [a]
eqList      = λ w.MyEq { eq = λ xs ys. case xs of
                                         [] -> case ys of
                                                 [] -> True
                                                 _  -> False
                                         (x' : xs') -> case ys of
                                                         [] -> False
                                                         (y' : ys') -> (eq w) x' y' && (eq (eqList w)) xs' ys',
                         neq = λ xs ys. not (eq (eqList w) xs ys) }
```

Figure 7. Naïve translation of eqList

```
eqList = λw . MyEq { eq = λ xs ys.
           case xs of
             [] -> case ys of
                     [] -> True
                     _  -> False
             (x' : xs') -> case ys of
                             (y' : ys') -> letrec h1 = λ w x y xs ys.
                                             case xs of
                                               [] -> case ys of
                                                       [] -> (eq w) x y
                                                       _  -> let z = (eq w) x y in False
                                               (x' : xs') -> case ys of
                                                               (y' : ys') -> (eq w) x y && h1 w x' y' xs' ys'
                                                               _  -> let z = (eq w) x y in False
                                           in h1 w x' y' xs' ys'
                             _ -> False
           (neq) = ... }
```

Figure 8. Supercompiled naïve translation of eqList

```
data E i e    = EId i | EAp e e
data Exp i    = Exp (E i (Exp i))

data Pretty a = Pretty { pp :: a -> String }

pE            = λ w1 w2.Pretty { pp = λ x1. case x1 of
                                             EId x2 -> w2.pp x2
                                             EAp x3 x4 -> "(" ++ (((pp w1) x3) ++ (" " ++ ( ((pp w1) x4) ++ ")"))) }
pExp          = λ w. Pretty { pp = λ x case x of
                                     Exp x' -> (pp (pE (pExp w) w)) x' }
```

Figure 9. pExp

```
h2    = λ w x. case x of
                 EId x1 -> w.pp x1
                 EAp x2 x3 -> "(" ++ (( case x2 of
                                          Exp x4 -> h2 w x4) ++ (" " ++ (( case x3 of
                                                                             Exp x5 -> h2 w x5) ++ ")")))
pExp = λ w. Pretty { pp = λ x. case x of
                                 Exp x' -> h2 w x' }
```

Figure 10. Supercompiled pExp

	Time (μs)		Allocations		Alloc Size	
Size	Before	After	Before	After	Before	After
1	3	3	4	2	20	10
5	4	3	12	2	60	10
10	4	3	22	2	110	10
50	9	5	102	2	510	10
100	21	8	202	2	1 010	10
500	91	20	1 002	2	5 010	10
1000	183	50	2 002	2	10 010	10
5000	927	240	10 002	2	50 010	10

Table 1. Time and allocation measurements for eqList

5.1 Benchmarks

We provide measurements for comparisons parametrized on the size of the input. We show that our positive supercompiler does remove dictionary allocations, which in turn saves both execution time and reduce memory allocation.

All measurements were performed on an idle machine running in an xterm. Each test was run 10 consecutive times and the best result was selected. The best result was selected since it must appear under the minimum of other activity of the operating system. We add a special memory allocation primitive to the runtime system that increases two counters for every time it is called: the number of calls and the total size allocated, and change the generated code to call this primitive from eqList, but not other portions of the code. This allows us to measure the allocations and runtime of eqList without any other interfering costs. The number of allocations and total allocation size remains constant over all runs.

The raw data for the time, size and allocation measurements are shown in Table 1. The number in the size column is the number of elements in the lists that are compared, so 500 means that two lists of 500 elements each were compared. The time column is number of microseconds from the *getrusage()* system call. The number of allocations and the number of words allocated by the eqList instance are displayed in each column.

The binary size increases from 238 984 bytes to 239 364 bytes for all input sizes. For lists of length 5 and more the runtimes are all faster, and in all cases both the number of allocations and the total size allocated decreased.

6. SAT

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.

Whenever we have mentioned this transformation we get the same response: "That shouldn't help". Whether it helps or not depends on how the compiler backend is designed: if lambda lifting is optional it might be a good idea to perform the static argument transformation; if the backend always lambda lifts there is no reason to perform the static argument transformation since the lambda lifter will undo the effects of SAT. However, measurements indicates that SAT is beneficial for GHC (Bolingbroke 2009).

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):

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

Performing SAT on this gives a version with a local definition without the f and z argument:

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

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. 2008a) is not going to give a speedup: the backend makes sure to lambda lift all functions before code generation.

6.1 Examples

Our supercompiler could not improve the first example in Section 2. We will now see how the modified supercompiler treat it differently, and jump straight to the equation that we named $(*)$:

$$\begin{array}{ll} \mathbf{case}\ ys\ \mathbf{of} & (*) \\ \quad [] \rightarrow [] & \\ \quad (x : xs) \rightarrow f'\ x : map\ f'\ xs & \end{array}$$

Up until this point there is no difference at all in the transformation, 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*). Since *f'* is the same in both, the recursive call will just put the same variable on the stack that the initial call did, and we can avoid that just like the static argument transformation does: make up a new function name (h_2) and apply it to *xs* to get:

$$\begin{array}{l} \mathbf{letrec}\ h_2 = \lambda xs.\ \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad [] \rightarrow [] \\ \qquad\qquad (x : xs) \rightarrow f'\ x : h_2\ xs \\ \mathbf{in}\ h_2\ xs \end{array}$$

Applying this transformation to the naïve translation of eqList (Figure 7) gives a similar result to the supercompiled eqList in Figure 8, except that the function h_1 would not have the dictionary *w* as a parameter: it would be a free variable. The same thing is true for h_2 in the pretty printer example (Figure 10).

6.2 SAT in Positive Supercompilation

By tweaking $\mathcal{D}_{app}(\ )$ it is possible to make positive supercompilation perform SAT. We need more function names: 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 11 shows the updated $\mathcal{D}_{app}(\ )$, with changes to alternative 1 and 4a. No other part of the algorithm needs modification, Figure 2 remains unchanged.

$$
\begin{array}{lcll}
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & h_{\overline{y}}\,\overline{z} & \text{if } \exists(\{h_i\},t)\in\rho\,.\,[\overline{z}/\overline{y}]t=\mathcal{R}\langle g\,\overline{e}\rangle \quad (1)\\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & \mathcal{R}\langle g\,\overline{e}\rangle & \text{if } \exists(\{h_i\},t)\in\rho\,.\,t\trianglelefteq\mathcal{R}\langle g\,\overline{e}\rangle \text{ and } \mathcal{R}\langle g\,\overline{e}\rangle\trianglelefteq t \quad (2)\\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & [\mathcal{D}[\![\overline{t}]\!]_{[],\mathcal{G},\rho}/\overline{y}]\mathcal{D}[\![t_g]\!]_{[],\mathcal{G},\rho} & \text{if } \exists(\{h_i\},t)\in\rho\,.\,t\trianglelefteq\mathcal{R}\langle g\,\overline{e}\rangle \quad (3)\\
\multicolumn{4}{l}{\quad \text{where } (t_g,\overline{t},\overline{y})=split(\mathcal{R}\langle g\,\overline{e}\rangle,t)}\\
\mathcal{D}_{app}(g,\overline{e})_{\mathcal{R},\mathcal{G},\rho} & = & \textbf{letrec}\ defs\ \textbf{in}\ h\,\overline{y} & \text{if } \exists(h=\lambda\overline{y}.e')\in defs \quad (4a)\\
 & & e' & \text{otherwise} \quad (4b)\\
\multicolumn{4}{l}{\quad \text{where } (g,e)\in\mathcal{G},\ e'=\mathcal{D}[\![\mathcal{R}\langle e\,\overline{e}\rangle]\!]_{[],\mathcal{G},\rho'},\ \overline{x}=fv(\mathcal{R}\langle g\,\overline{e}\rangle),\ \rho'=\rho\cup(\{h_{\overline{y}}|\overline{y}\subseteq\overline{x}\},\mathcal{R}\langle g\,\overline{e}\rangle)}\\
\multicolumn{4}{l}{\quad\quad defs=\{h_{\overline{y}}=\lambda\overline{y}.e'|\overline{y}\subseteq\overline{x}\},\ h_i \text{ fresh}}
\end{array}
$$

Figure 11. Modified driving of applications

The condition for alternative 4a is true whenever alternative 1 was selected at least once during the transformation of *e'*. 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.

If a function has multiple recursive calls, with different static arguments, this transformation will create a group of mutually recursive functions. This group of mutually recursive functions will suffer from code duplication. It is also possible to create recursion patterns that alternates between these functions, thereby destroying locality properties that an ordinary supercompiled function would have.

6.3 Correctness

We can prove that this modified algorithm terminates by updating our old termination proof for the modified parts of the algorithm, and this can be shown for both call-by-name and call-by-value.

Lemma 6.1 (Totality)**.** *For all well-typed expressions e, $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{G},\rho}$ is matched by a unique rule in Figure 2.*

Proposition 6.2 (Termination)**.** *The driving algorithm $\mathcal{D}[\![\ ]\!]$ terminates for all well-typed inputs.*

We can not use the same proof technique for preservation of semantics for this case: we have a primitive letrec in the language but the GDSOS format used by Sands (1997) requires that there are no free (ordinary) variables in the metaterms. We intend to work on this this problem in our future work.

7. Related Work

There are three orthogonal directions for related work to what we just presented: supercompilation, SAT, and techniques for optimizing type classes.

7.1 Supercompilation

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

The *positive supercompiler* (Sørensen et al. 1996) is a variant which only propagates positive information, such as equalities. The propagation is done by unification and the work highlights how similar deforestation and positive supercompilation really are. We have previously investigated the theoretical foundations for positive supercompilation for strict languages (Jonsson and Nordlander 2009). Narrowing (Albert and Vidal 2001) is the functional logic programming community equivalent of positive supercompilation but formulated as a term rewriting system. They also deal with nondeterminism from backtracking, which makes the algorithm more complicated.

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

More recently, Mitchell and Runciman (2008) have worked on supercompiling Haskell. Their algorithm is closely related to our supercompiler, but they limit their work to call-by-name. They report runtime reductions of up to 55% when their supercompiler is used in conjunction with GHC. Our current work should be seen as complimentary to their work: their work is about making supercompilation a feasible optimization technique for real world programs and our work here is about characterizing when supercompilation is beneficial.

7.2 Static Argument Transformation

The static argument transformation was first described by Santos (1995), including measurements from an implementation in GHC that 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 GHC.

The idea to do SAT is obviously not new, but we have not seen it reported in the context of supercompilation earlier. Considering the similarity between the transformations it is not surprising that a slightly tweaked positive supercompilation can subsume SAT. The problem of characterizing when SAT is beneficial remains: what we have shown is that given a positive supercompiler it is possible and cheap to modify it to do SAT as well.

7.3 Optimizing Type Classes

An early contribution on optimization techniques for type classes was by Augustsson (1993) where he described several straightforward techniques that Haskell compilers implemented at the time, and if these optimizations were combined they could have a significant impact on performance.

Augustsson describes exactly the same problem that we mention in the introduction: how a new dictionary is created and a single component is selected, effectively throwing away the rest of the record. A clever representation of dictionaries is used to avoid this and reduce the number of memory accesses on the G-machine (Johnsson 1987). We have seen several examples of how our supercompiler can remove this problem completely, so this optimization should not be necessary.

Type classes that have superclasses normally contain pointers to their superclasses. This can be flattened to a single dictionary thereby reducing the number of indirections. In cases where the dictionaries have to remain in code, such as polymorphic recursion, this is a good idea that our supercompiler can not address: the optimization is to change the representation of data structures to speed up access to them.

Augustsson describes both "simple" and "real" partial evaluation, which is very similar to what our supercompiler achieves: functions specialized with respect to their dictionaries. Our tests on eqList was for the polymorphic case, but if the type is known statically it will produce a specialized version of the function that does not need a dictionary.

Strict data types (the "!" annotation available in GHC for example) is one more thing our supercompiler can not accomplish: this is about changing the semantics of a data type, and must be done by the programmer. Augustsson also suggests the use of compiler pragmas, something we briefly discussed in Section 5.

Jones (1995) used partial evaluation to remove the run-time passing of dictionaries by generating specialized versions of overloaded functions, and somewhat surprisingly this led to a reduction of size of the compiled programs. If all the type information is available our supercompiler will specialize overloaded functions, just like Jones does. While his work discusses polymorphic recursion it does not provide an explicit solution to the problem since neither Haskell nor Gofer allowed such programs at the time. We can not remove the dictionaries in the presence of polymorphic recursion, but our supercompiler is guaranteed to terminate.

8. Conclusions

We have shown how a supercompiler can automatically remove the dictionary allocations from naïve translations of lists and pretty printers. We tried supercompiling equality instances for various kinds of trees as well, and had the same results there: dictionary allocations were removed. Our adjustment to the folding mechanism to get the supercompiler to do SAT is not something we have seen mentioned elsewhere, although SAT is a well known transformation.

8.1 Future Work

We are currently working on extending positive supercompilation for strict languages to overcome the limitation in the example with pExp in Section 5.

The work on Supero by Mitchell and Runciman (2008) shows that there are open problems for supercompiling large Haskell programs. These problems are mainly relating to speed, both of the compiler, and of the transformed program. When they profiled Supero, they found that the majority of the time was spent in the homeomorphic embedding test. We have just shown that supercompiling Eq-instances is feasible and beneficial, for our immediate future work we need to scale this to whole programs. MLTon (Weeks 2006) have successfully compiled whole programs up to 100 000 lines, so more investigation on performance issues with supercompilation is warranted.

The lack of proof for semantic preservation of the modified supercompiler mentioned in Section 6.3 is an obvious candidate for future work. We believe that our algorithm does preserve the semantics, it is merely a clash with the preconditions set by the GDSOS format used by the improvement theory. Whether we should try to weaken those preconditions or moving away from the improvement theory is still an open question that needs more investigation.

Characterizing when it is beneficial to use the modified supercompiler is something that would be very interesting to do, but in practice this requires an implementation of a supercompiler in GHC or some other compiler where this transformation could be useful. Just supercompiling System F_c (Sulzmann et al. 2007), the intermediate language of GHC, is in itself a challenging problem

Acknowledgments

The authors would like to thank Thomas Hallgren for the pretty printer example, Max Bolingbroke for explaining the static argument transformation and Simon Marlow for answering questions about GHC. Germán Vidal made insightful comments that paved the way for the simplification of the algorithm. We would also like to thank Viktor Leijon for providing useful comments that helped improve the presentation and contents.

References

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

L. Augustsson. Implementing haskell overloading. In *FPCA*, pages 65–73, 1993.

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

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

T. Hallgren. Haskell tools from the programatica project. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 103–106, New York, NY, USA, 2003. ACM. ISBN 1-58113-758-3.

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

T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1987.

M. P. Jones. Dictionary-free overloading by partial evaluation. *Lisp and Symbolic Computation*, 8(3):229–248, 1995.

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 sympos ium on Principles of programming languages*, 2009.

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.

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.

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

J. Nordlander, M. Carlsson, and A.J. Gill. Unrestricted pure call-by-value recursion. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 23–34, New York, NY, USA, 2008b. ACM. ISBN 978-1-60558-062-3.

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. Call-pattern specialisation for haskell 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 327–337. ACM, 2007. ISBN 978-1-59593-815-2. URL `http://doi.acm.org/10.1145/1291151.1291200`.

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

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

D. Sands. From SOS rules to proof principles: An operational 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.

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. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Don nelly. 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. doi: http://doi.acm.org/10.1145/1190315.1190324.

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

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

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

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

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

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

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

S. Weeks. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159877. http://mlton.org/pages/References/attachments/060916-mlton.pdf.