Local Driving in Higher-Order Positive Supercompilation via the Ω-theorem

G.W. Hamilton[1] and M.H. Sørensen[2]

[1] School of Computing and Lero, Dublin City University, Ireland
hamilton@computing.dcu.ie
[2] Formalit, Store Heddinge, Denmark
mhs@formalit.dk

Abstract

A program transformation technique should terminate, return efficient output programs and be efficient itself. These requirements are mutually conflicting, so a balance must be sought between definite termination and possible efficiency.

For positive supercompilation [17], ensuring termination requires memoisation of expressions, and these are subsequently used to determine when to perform generalization and folding [16]. For a first-order language, every infinite sequence of transformation steps must include function unfolding, so it is sufficient to memoise only those expressions immediately prior to a function unfolding step.

However, for a higher-order language, it is possible for an expression to have an infinite sequence of transformation steps which do not include function unfolding, so memoisation prior to a function unfolding step is not sufficient by itself to ensure termination. But memoising additional expressions is expensive during transformation and may lead to less efficient output programs due to auxiliary functions. This additional memoisation may happen explicitly during transformation or implicitly via a pre-processing transformation as outlined in previous work by the first author [5].

We introduce a new technique for local driving in higher-order positive supercompilation which obliviates the need for memoising other expressions than function unfolding steps, thereby improving efficiency of both the transformation and the generated programs. We exploit the fact, due to the second author in the setting of type-free λ-calculus [20] known as the Ω-theorem, that every expression with an infinite sequence of transformation steps not involving function unfolding must have the term $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$ embedded within it in a certain sense. The technique has proven useful on a host of examples.

1 Introduction

Supercompilation is a program transformation technique for functional languages which can be used for program specialization, removal of intermediate data structures, as well as other optimizations. Supercompilation was devised by Valentin F. Turchin in the early 1970s in the context of the language Refal [18, 19] and studied by numerous researchers in the USSR. The original ideas had a broad scope with, for instance, a bearing on cybernetics. A specific variant of the transformation techinque was later popularized in the setting of a more traditional functional language in a form known as *positive supercompilation* [14, 17]. More recently, it has been generalized to higher-order languages in several new lines of work [13, 2, 7, 8, 9, 5].

Ensuring termination of positive supercompilation requires the memoisation of expressions and using these memoised expressions to determine when to perform generalization and folding. Positive supercompilation was originally formulated for a first-order language, so it was sufficient to memoise only the expressions immediately prior to function unfolding to ensure termination, since in any infinite sequence of transformation steps there must be an unfolding.

However, memoising expressions immediately prior to function unfolding is not sufficient to ensure termination in a higher-order setting. For example, consider the following λ-expression:

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

When this expression is transformed there will be an infinite sequence of transformation steps without any unfolding (the same expression Ω is encountered in each step). Although the expression would not be accepted by most type checkers, there are examples of expressions which would be accepted by a type checker and which have an infinite sequence of transformation steps without any unfolding. For example, consider the following expression in the context of a datatype **data** D = F (D $\rightarrow$ D):

$$(\lambda f.f\ (F\ (\lambda x.f\ x\ x))\ (F\ (\lambda x.f\ x\ x)))\ (\lambda y.\mathbf{case}\ y\ \mathbf{of}\ F\ g.g)$$

This expression also leads to an infinite sequence of transformation steps with no unfoldings.

To avoid non-termination, some formulations of positive supercompilation for a higher-order language memoise *all* expressions [13, 2], or at least a substantial subset of them [7, 8, 9, 5]. However, this may be a costly operation during transformation and may result in less optimized output programs (that may actually be less efficient than the input programs).

In this paper, we therefore introduce a new technique for local driving in higher-order positive supercompilation which obliviates the need for memoising other expressions than function unfolding steps, thereby improving efficiency of the generated programs as well as of the transformer itself. We exploit the fact, known from type-free λ-calculus [20], that every expression with an infinite sequence of transformation steps not involving function unfolding must have the term $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$ embedded within it in a certain sense. We can see that this is, indeed, the case for the above examples.

The core of the technique is to check each expression encountered during transformation not involving function unfolding to see whether Ω is embedded. If so, the expression is generalized to remove the source of infinite transformation. We prove that the resulting transformation will always terminate, using the setting of abstract program transformers [15]. We also demonstrate the usefulness of the technique on a range of examples.

The paper reports experimental work in progress. For instance, the relation between our input and output programs is not formalized in terms of a model for execution cost.

The remainder of this paper is structured as follows. In Section 2, we describe the higher-order language over which the transformations are defined. In Section 3, we recall the setting of abstract program transformers. In Section 4, we give our formulation of the positive supercompilation algorithm using the mentioned technique for local driving and prove that it terminates. Section 5 concludes and considers related work.

2 Language

We describe the higher-order functional language which will be used throughout the paper.

Definition 2.1 (Syntax)**.**

Δ	$::=$	$f_1 = e_1 \ldots f_n = e_n$	*Definitions*
e	$::=$	x	*Variable*
	$\mid$	f	*Function Call*
	$\mid$	$c\ e_1 \ldots e_n$	*Constructor*
	$\mid$	**case** e_0 **of** $p_1 \rightarrow e_1;\ \cdots p_n \rightarrow e_n;$	*Case Expression*
	$\mid$	$\lambda x\ .e$	*λ-Abstraction*
	$\mid$	$e_0\ e_1$	*Application*
	$\mid$	**let** $x = e_0$ **in** e_1	*Let Expression*
p	$::=$	$c\ x_1 \ldots x_n$	*Pattern*

Informally speaking, a program is a sequence of function definitions together with an expression that is evaluated in the context of the function definitions. An expression can be a variable, function call, constructor expression, **case**, λ-abstraction, application or **let**.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. We use [] and $(x : xs)$ as abbreviations for *Nil* and *Cons*. The patterns in a **case** expression must be non-overlapping and exhaustive, and a pattern must not refer to the same variable twice. The expressions in function definitions must refer only to functions defined in the program. It is assumed that erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x\ .e)$ **of** $p_1 \rightarrow e_1;\ \cdots p_n \rightarrow e_n;$ do not occur.

Variables introduced by λ-abstraction, **let** or **case** patterns are *bound*; all other variables are *free*. We use $FV(e)$ to denote the free variables of expression e. We identify expressions that differ only in the names of bound variables; that is, we think of expressions as equivalence classes of a relation that equals syntactic expressions differing only in the choice of names for bound variables. In any situation we assume that names for bound variables are chosen so as to avoid name capture. We also chose different names for different bound variables within an expression.

In the context of any specific program, the number of function symbols is finite. For a case-expression **case** e_0 **of** $p_1 \rightarrow e_1;\ \cdots p_n \rightarrow e_n;$ the sequence $p_1, \ldots, p_n$ is called the *brand* of the case-expression. In the context of any specific program, there are only finitely many different brands of case-expressions. When we speak about "arbitrary" expressions it will always be in the context of a specific program.

Definition 2.2 (Reduction Context)**.** A reduction context R is an expression containing a single hole [], which can have one of the following forms:

$$R ::= R\ e \mid \textbf{case}\ R\ \textbf{of}\ p_1 \rightarrow e_1;\ \ldots\ p_n \rightarrow e_n;\ \mid []$$

By $R[e]$ we denote the result of replacing the hole in R by the expression e.

Definition 2.3 (Substitution)**.** *A partial map* $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ from variables to expressions is called a *substitution*. By $\mathrm{dom}(\theta) = \{x_1, \ldots, x_n\}$ we denote the *domain* of the map. If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the variables $x_1, \ldots, x_n$, respectively, in e. We write $e \leq e'$ if $e' = e\theta$ for some substitution θ.

Definition 2.4 (Operational Semantics)**.** We define a one-step reduction $\rightsquigarrow$ on expressions (in the context of a program Δ) as shown below, comprising function unfolding, pattern matching,

β-reduction and **let**-unfolding.

$$
\begin{array}{lcll}
R[f] & \rightsquigarrow & R[e] & (\text{where } f = e \in \Delta) \\
R[(\lambda x.e_0)\ e_1] & \rightsquigarrow & R[e_0\{x \mapsto e_1\}] & \\
R[\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ \ldots\ (c\ x_1 \ldots x_n) : e; \ldots] & \rightsquigarrow & R[e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}] & \\
R[\textbf{let}\ x = e_0\ \textbf{in}\ e_1] & \rightarrow & R[e_1\{x \mapsto e_0\}] &
\end{array}
$$

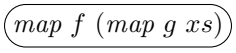
3 Abstract Program Transformers

We recall how program transformers may be viewed as maps that manipulate certain trees. Our exposition is informal; a more detailed and rigorous presentation can be found in [15]. Consider the well-known *map* function which maps a function to the elements of a list.

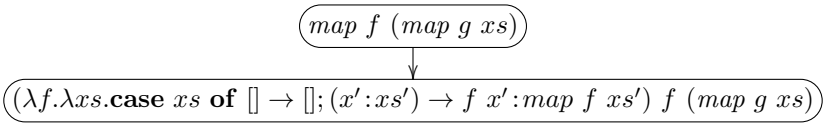
$$map = \lambda f.\lambda xs.\textbf{case}\ xs\ \textbf{of}\ [] \rightarrow []\ ;\ (x' : xs') \rightarrow f\ x' : map\ f\ xs'$$

Suppose we want to apply *two* functions to a list. A simple and elegant way is to use the expression $map\ f\ (map\ g\ xs)$. However, this expression is inefficient since it traverses xs twice. We now illustrate a standard transformation obtaining a more efficient method.[1]

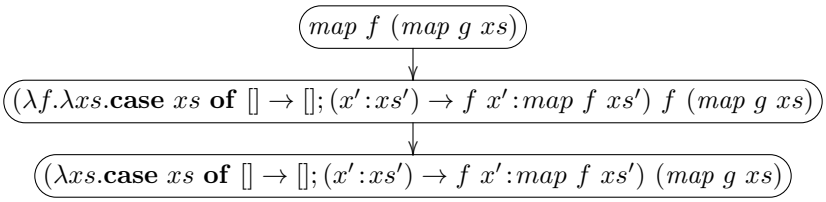
We begin with a tree whose single node is labeled with $map\ f\ (map\ g\ xs)$:



Transformation mimicks evaluation as much as possible; by an *unfold* step which replaces the outer call to *map*, a new expressions is added as child:

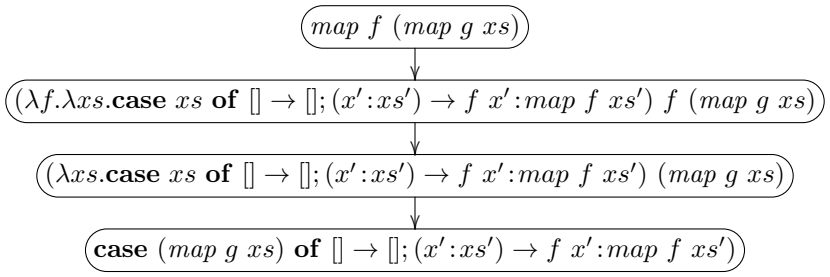


Unfold steps are similar to evaluation steps except that the former apply to expressions with variables. Performing a β-reduction yields another child:

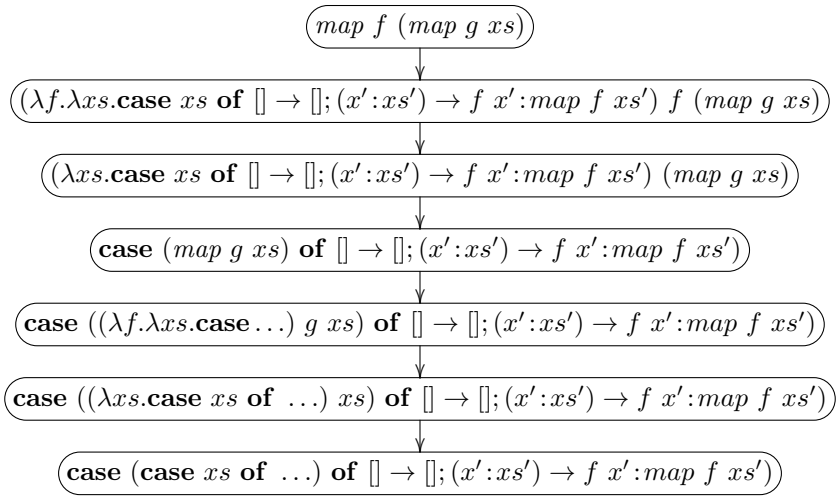


[1] Note that in this example we use f and g as variables, whereas elsewhere f ranges over names of functions defined in a program.

Another β-reduction yields yet another child:



Now we unfold the inner call to *map* followed by another two β-reductions:



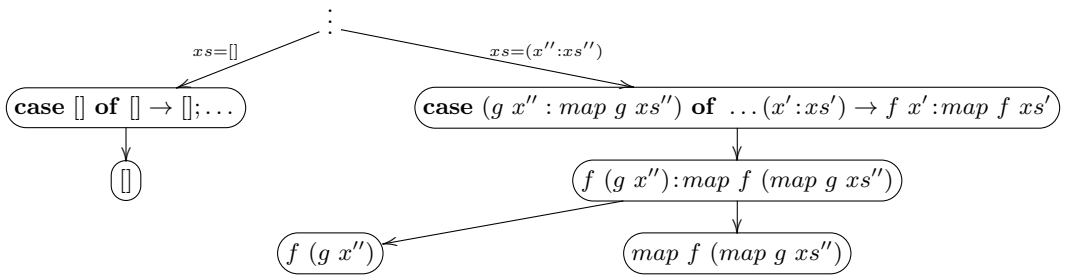
The expression in the leaf is

case (**case** xs **of** $[] \to []$; $(x'' : xs'') \to g\ x'' : map\ g\ xs''$) **of** $[] \to []; (x' : xs') \to f\ x' : map\ f\ xs'$;

At this point evaluation is stuck, we really need to know what xs is to make progress. In this case we move the outer case to the branches of the inner like this:

case xs **of**
$[]$ → **case** $[]$ **of** $[] \to []$; $(x' : xs') \to f\ x' : map\ f\ xs'$;
$(x'' : xs'')$ → **case** $(g\ x'' : map\ g\ xs'')$ **of** $[] \to []$; $(x' : xs') \to f\ x' : map\ f\ xs'$;

We then add the two inner (lower) case-expressions as children to transform them further and label the arrows with the corresponding patterns. Thus the tree proceeds as follows:



The left branch just adds another child by a single case-evaluation step. The right branch first evaluates the case-expression and then adds a branch for each of the two parts of the constructor-expression The expression $f\ (g\ x'')$ is all variables, so no progress can be made. The expression in the rightmost child is a renaming of the expression in the root we began with; that is, the two expressions are identical up to choice of free variable names. No further processing of such a node is required.

The tree is now *closed* in the sense that each leaf expression either is a renaming of an ancestor's expression, or contains all variables or is a 0-ary constructor. A closed tree is a representation of all possible computations with the expression e in the root, where branches in the tree (labelled with patterns) correspond to different run-time values for the free variables of e (that match the patterns). In the above tree, computation starts in the root with values for f, g, and xs, and then branches to one of the successor states depending on the shape of xs. Assuming xs is $[]$, the value $[]$ is returned. But if xs has form $(x'':xs'')$, the constructor : is emitted and control is passed to the two states corresponding to nodes labeled $f\ (g\ x'')$ and $map\ f\ (map\ g\ xs'')$, where the latter returns control to the root.

To construct a new expression and corresponding program from a finite, closed tree, we proceed roughly as follows. First remove any nodes that do not involve branching and do not contain an expression that is later encountered again (up to renaming). Then every node with free variables $x_1, \ldots x_n$ is viewed as a call $m\ x_1, \ldots x_n$ to a new function m, whose definition begins with $m = \lambda x_1 \ldots \lambda x_n \cdots$. The children of the node define the right hand side of the new function definition which, in the case of multiple children, will contain a constructor, a let-expression, or a case-expression.

For example, from the above tree, the expression $m\ f\ g\ xs$ and program

$$m = \lambda f.\lambda g.\lambda xs.\textbf{case}\ xs\ \textbf{of}\ [] \to []; \ (x' : xs') \to f\ (g\ x)' : m\ f\ g\ xs'$$

can be extracted. The new expression is more efficient than the original, since the new one traverses xs only once.

In the above transformation we ended up with a finite closed tree. Often, special *generalization* steps must be performed to ensure that this situation is eventually encountered. Such steps replace a subtree (e.g. a leaf) with expression e by a new node with expression **let** $x = e_0$ **in** e_1 such that $e_1\{x \mapsto e_0\} = e$. The transformation adds branches for e_0 and e_1 which are then processed independently.

In conclusion, a program transformer is a map from trees to trees, expressing one step of transformation.

Definition 3.1. T_∞ is the set of all trees labelled with expressions, and T is the set of all finite trees labelled with expressions. A *singleton* tree is a tree with no other nodes than the root. An *abstract program transformer* (for brevity also called an *apt*) is a map $M : T \to T$.

An apt only computes a *single* step of transformation: it maps some tree to a new tree by performing, e.g., an unfolding step. Hence, the sequences of trees in the example in this section could be computed by *iterated* application of some apt. How do we express that no more transformation steps will happen, i.e., that the apt M has produced its final result? In this case, M returns its argument tree unchanged, i.e., $M(t) = t$.

Definition 3.2.

1. An apt M *terminates on* $t \in T$ if $M^i(t) = M^{i+1}(t)$ for some $i \in \mathbb{N}$.[2]
2. An apt M *terminates* if M terminates on all singletons $t \in T$.

[2]For $f : A \to A$, $f^0(a) = a, f^{i+1}(a) = f^i(f(a))$.

4 Higher-order Positive Supercompilation

We now present higher-order positive supercompilation as an abstract program transformer. Whereas the preceding sections have been largely informal leaving rigourous details to e.g. [15], the present section attempts to be precise. However, as our main concern is to prove termination for the technique, we focus on how to construct the trees and make them finite, not on recovering new programs from the trees.

The first subsection presents the driving operation. The next two subsections introduce the generalization operations, covering *when* to generalize and *how* to generalize, respectively. The fourth subsection brings together the pieces in an algortihm for higher-order positive supercompilation. The fifth subsection adds our new technique for local driving to higher-order positive supercompilation—this is the main contribution of the paper.

4.1 Driving

When we perform unfolding steps, we instantiate variables to patterns, e.g. *xs* to (*x'*:*xs'*). To avoid confusion of variables, when instantiating a variable to a pattern we are free to use whatever fresh variable names in the pattern we like, as long as we use the same names in the corresponding right hand side of the function definition. The following definition formalizes what it means to be fresh.

Definition 4.1.

1. *The* yield *of a substitution* θ *is:* $\text{yield}(\theta) = \bigcup\{FV(x\theta) \mid x \in \text{dom}(\theta)\}$.
2. *A substitution* θ *is* free for e *if* $\text{yield}(\theta) \cap FV(e) = \emptyset$.

The crucial property of a substitution θ which is free for an expression e is that the variables in the range of θ do not occur already in e. We will always choose substitutions that instantiate a variable to a pattern free for the expression containing the variable to be instantiated.

Unfolding steps add children to leaf nodes. The essence in defining the unfolding step is to define how the expressions in the new children are computed from the leaf's expression. This computation is formalized by the following relation $\Rightarrow$.

Definition 4.2. The relation $e \Rightarrow e'$ is defined as follows. The expression on the left hand side of $\Rightarrow$ must be free for the substitutions used on right hand sides.

$$R[f] \Rightarrow R[e] \text{ where } f = e \in \Delta \quad (1)$$

$$R[(\lambda x.e_0)\ e_1] \Rightarrow R[e_0\{x \mapsto e_1\}] \quad (2)$$

$$R[\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ \ldots (c\ x_1 \ldots x_n) : e; \ldots] \Rightarrow R[e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}] \quad (3)$$

$$R[\textbf{case}\ x\ \textbf{of}\ p_1 : e_1; \ldots; p_k : e_k] \Rightarrow R[e_i]\{x \mapsto p_i\} \quad (4)$$

$$R[x\ e] \Rightarrow e, R[y] \quad (5)$$

$$c(e_1, \ldots, e_n) \Rightarrow e_i \quad (6)$$

$$\lambda x \to e \Rightarrow e \quad (7)$$

$$R[\textbf{let}\ x = e_0\ \textbf{in}\ e_1] \Rightarrow e_0, R[e_1] \quad (8)$$

Rules (5) and (8) state that the left hand side is related by $\Rightarrow$ to two different terms; for instance **let** $x = y$ **in** $z \Rightarrow y$ and **let** $x = y$ **in** $z \Rightarrow z$. The fresh variable y in clause (5) reflects the idea that we pull out the stuck application $x\ e$ from the context and reduce e and R separately as in **let** $y = x\ e$ **in** $R[y]$.

The new relation generalizes $\rightsquigarrow$ in several ways. First of all, the reduction for let-expressions expresses the semantics of generalizations: that we are keeping things apart.

Next, note that the rules propagate to the arguments of constructors (including λ); that is, we do not stop at weak head normal forms.

Finally, transformation works on expressions with free variables. Stuck case expressions are handled by propagating unifications representing the assumed outcome of pattern matching—notice the substitution $\{x_i := p_i\}$ in the fourth rule.

The unfolding operation is called *driving* and is defined in Figure 1 together with some generalization operations introduced in the next subsections. For a node α in a tree labeled with expressions, we use $t(\alpha)$ to denote the expression that labels α.

4.2 Generalization: when

Next we formulate the generalization operations used in higher-order positive supercompilation. In this subsection we present the technique which decides *when* to generalize; the next subsection presents the *how*. The aim of *when* and *how* combined is to balance termination of the transformer with efficiency of output programs.

To ensure the termination, generalization is performed when an expression is encountered which contains a *homeomorphic embedding* of a memoised expression. Variants of this technique have been used to ensure termination within term rewriting [3], positive supercompilation [16, 15], partial evaluation [11] and partial deduction [1, 10] (see also [4]). These papers also contain references to the earlier classical works of Higman, Kruskal and Nash-Williams.

Later it has been used for higher-order positive supercompilation—see [13, 2, 6, 12, 7, 8, 9, 5]. In some of these works two expressions must be coupled at the outer-most level to avoid the split-operation; but we will need that operation anyway. Also, bound variables must be match up, but this is less important here, since we will compare fewer expressions to their ancestors.

Definition 4.3 (Homeomorphic Embedding)**.** For expressions e, e' define $e \trianglelefteq e'$as follows.

$$f \trianglelefteq f \qquad\qquad x \trianglelefteq y$$

$$\frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq e'_i}{c\ e_1 \ldots e_n \trianglelefteq c\ e'_1 \ldots e'_n} \qquad\qquad \frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq c\ e_1 \ldots e_n}$$

$$\frac{e \trianglelefteq e'}{\lambda x.e \trianglelefteq \lambda y.e'} \qquad\qquad \frac{e \trianglelefteq e'}{e \trianglelefteq \lambda x.e'}$$

$$\frac{\forall i \in \{0, 1\}.e_i \trianglelefteq e'_i}{e_0\ e_1 \trianglelefteq e'_0\ e'_1} \qquad\qquad \frac{\exists i \in \{0, 1\}.e \trianglelefteq e_i}{e \trianglelefteq e_0\ e_1}$$

$$\frac{\forall i \in \{0 \ldots n\}.e_i \trianglelefteq e'_i \text{ and } p_i \text{ is a renaming of } p'_i}{\textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1; \ldots; p_n \to e_n \trianglelefteq \textbf{case}\ e'_0\ \textbf{of}\ p'_1 \to e'_1; \ldots; p'_n \to e'_n}$$

$$\frac{\exists i \in \{0 \ldots n\}.e \trianglelefteq e_i}{e \trianglelefteq \textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1; \ldots; p_n \to e_n}$$

An expression is embedded within another by this relation if either *diving* or *coupling* can be performed. Diving occurs when an expression is embedded in a sub-expression of another expression, and coupling occurs when two expressions have the same top-level construct and all the corresponding sub-expressions of the two constructs are embedded.

Example 4.4. Some examples of homeomorphic embedding are as follows:

1.	$f\ (g\ x) \trianglelefteq f\ (g\ y)$	6.	$f\ (g\ x) \not\trianglelefteq\ g\ (f\ y)$
2.	$f\ (h\ x) \trianglelefteq f\ (g\ (h\ y))$	7.	$g\ (h\ x) \trianglelefteq\ \ f\ (g\ (h\ y))$
3.	$f\ x\ y \trianglelefteq f\ z\ z$	8.	$f\ z\ z \trianglelefteq\ \ f\ x\ y$
4.	$\lambda x.x \trianglelefteq \lambda y.y$	9.	$\lambda x.x \trianglelefteq\ \ \lambda y.x$
5.	$\lambda x.f\ y \trianglelefteq \lambda x.f\ (g\ y)$	10.	$\lambda x.f\ x \trianglelefteq \lambda x.f\ (g\ x)$

Theorem 4.5. *The relation $\trianglelefteq$ is a well-quasi order; that is, for any infinite sequence of expressions $e_1, e_2, \ldots$ there must be i, j with $e_i \trianglelefteq e_j$.*

Proof. We can view expressions as first-order terms with two functions for abstraction and application $\mathrm{Lam}(e)$ and $\mathrm{App}(e, e')$. Similarly, each case-expression **case** e_0 **of** $p_1 \to e_1;\ \cdots p_n \to e_n$; can be viewed as constructed from an $n+1$-ary function Case (different function for each brand of case-expression).

Replacing all free and bound variables by a common 0-ary constructor, results in a first-order term over a finite set of symbols and the relation $\trianglelefteq$ corresponds to the standard homeomorphic embedding on this first-order representation. The result follows from the first-order case. □

A way of using the homeomorphic embedding relation to decide whether to drive a given leaf or generalize is as follows: if the leaf has an ancestor whose expression is embedded in the leaf's expression, then we should generalize; if not, we should drive.

The rationale behind using the homeomorphic embedding relation in this way is that in any infinite sequence $e_0, e_1, \ldots$ of expressions, there *definitely* are $i < j$ with $e_i \trianglelefteq e_j$. Thus, if driving is stopped at any node with an expression in which an ancestor's expression is embedded, driving cannot construct an infinite branch. Conversely, if $e_i \trianglelefteq e_j$ then all the subexpressions of e_i are present in e_j embedded in extra subexpressions. This suggests that e_j *might* arise from e_i by some infinitely continuing system, so driving is stopped for a good reason.

But in fact, this reason is not always good. In order to avoid premature generalization, it is desirable that the embedding relations be as small as possible. Therefore, it is customary to consider restrictions of it, either by explicitly restricting the relation, or by comparing fewer expressions. In this paper, we chose the latter approach.

The homeomorphic embedding relation is defined on expressions without the let-construct. We will start out with an expression without let-expressions. Whenever transformation introduces a new node with a let-expression (by generalization), we immediately drive the node, resulting in children without let-expressions. When a node is compared to ancestors we do not compare it to those with let-expressions.

In conclusion, given a tree we may drive a leaf provided no relevant ancestor has an expression which is homeomorphically embedded in the leaf's expression. In the next subsection we present the generalization operations to be performed when some relevant ancestor *does* have an expression which is homeomorphically embedded in the leaf's expression.

4.3 Generalization: how

In generalization steps one compares two expressions and extracts common structure.

Definition 4.6 (Computing generalization)**.**

$$x \sqcap x = (x, \{\}, \{\})$$

$$f \sqcap f = (f, \{\}, \{\})$$

$$\begin{array}{ll}(c\ e_1 \ldots e_n) \sqcap (c\ e'_1 \ldots e'_n) = & (c\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i) \\ & \text{where} \\ & \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i\end{array}$$

$$\begin{array}{ll}(\lambda x.e_0) \sqcap (\lambda x.e'_0) = & (\lambda x.e^g_0, \theta_0, \theta'_0) \\ & \text{where} \\ & (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \text{ and } x \notin \text{yield}(\theta_0) \cup \text{yield}(\theta'_0)\end{array}$$

$$\begin{array}{ll}(e_0\ e_1) \sqcap (e'_0\ e'_1) = & (e^g_0\ e^g_1, \theta_0 \cup \theta_1, \theta'_0 \cup \theta'_1) \\ & \text{where} \\ & (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \\ & (e^g_1, \theta_1, \theta'_1) = e_1 \sqcap e'_1\end{array}$$

$$\begin{array}{l}(\textbf{case}\ e_0\ \textbf{of}\ p_1 \to e_1; \ldots; p_n \to e_n) \sqcap (\textbf{case}\ e'_0\ \textbf{of}\ p_1 \to e'_1; \ldots; p_n \to e'_n) = \\ \quad (\textbf{case}\ e^g_0\ \textbf{of}\ p_1 \to e^g_1; \ldots; p_n \to e^g_n, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta'_i) \\ \quad \text{where} \\ \quad (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap e'_0 \\ \quad \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \text{ and } FV(p_i) \cap (\text{yield}(\theta_i) \cup \text{yield}(\theta'_i)) = \emptyset\end{array}$$

$$e \sqcap e' = (x, \{x \mapsto e\}, \{x \mapsto e'\}) \qquad \text{in all other cases } (x \text{ is fresh})$$

The following rewrite rule is exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions that were previously given different names:

$$(e, \theta \cup \{x \mapsto e', x' \mapsto e'\}, \theta' \cup \{x \mapsto e'', x' \mapsto e''\}) \Rightarrow (e\{x \mapsto x'\}, \theta \cup \{x' \mapsto e'\}, \theta \cup \{x' \mapsto e''\})$$

We say that e_1 and e_2 are *incommensurable,* $e_1 \leftrightarrow e_2$, if $e_1 \sqcap e_2 = (x, \theta_1, \theta_2)$.

Within these rules, if both expressions have the same top-level construct, this is made the top-level construct of the resulting generalized expression, and the corresponding sub-expressions within the construct are then generalized (for λ-abstractions and case-expressions it must be ensured that the substitutions do not lead to variable capture). Otherwise, both expressions are replaced by the same fresh variable. It is assumed that the new variables introduced are all different and distinct from the original program variables.

The results of applying this generalization to items 1-5 in Example 4.4 are as follows:

1. $(f\ g\ v, \{v \mapsto x\}, \{v \mapsto y\})$
2. $(f\ v, \{v \mapsto h\ x\}, \{v \mapsto g\ (h\ y)\})$
3. $(f\ v_1\ v_2, \{v_1 \mapsto x, v_2 \mapsto y\}, \{v_1 \mapsto z, v_2 \mapsto z\})$
4. $(\lambda x.x, \{\}, \{\})$
5. $(\lambda x.f\ v, \{v \mapsto y\}, \{v \mapsto g\ y\})$

Example 8 is similar to Example 3; in the remaining cases, the generalization of e_1 and e_2 has form $(v, \{v \mapsto e_1\}, \{v \mapsto e_2\})$.

Positive supercompilation uses two types of generalization step: *abstract* and *split;* the former type, in turn, comes in two variants, *upwards* abstract and *downwards* abstract. All three types of steps may be invoked when the expression of a leaf node has a relevant ancestor's expression embedded.

In upwards abstraction we replace the tree whose root is the *ancestor* by a single new node labeled with a new expression which captures the common structure of the leaf and ancestor expressions. This common structure is computed by the most specific generalization operation.

In case the leaf expression is an instance of the ancestor expression, the msg of the two expressions is the same as the ancestor expression. Hence, it does not make sense to attempt to extract some common structure at the ancestor and continue with that: this structure is the ancestor itself. However, we can replace the *leaf* node by a new node with an expression capturing the common structure. This is what a downwards abstract step does. For instance, if the leaf expression is $f\,(x' : xs')$ and the ancestor expression is $f\,xs$, we can replace the leaf node by a node with expression $\mathbf{let}\ xs{=}x' : xs'\ \mathbf{in}\ f\,xs$. By driving, this node will receive two children labeled $x' : xs'$ and $f\,xs$; since the latter node is now a renaming of the ancestor's expression, no further processing of it is required.

In some cases, the expression of a leaf node may have an ancestor's expression embedded, and yet the two expressions have no common structure in the sense of msg's, i.e., the expressions are incommensurable (their msg is a variable). In this case, performing an abstract step—whether upwards or downwards—would not make any progress towards termination of the supercompilation process. For instance, we might have a leaf with expression $(\lambda x.e_0)\ e_1$ and an ancestor with expression $\lambda x.e_0$, so their msg is a variable. Therefore, applying an abstract step (upwards or downwards) would replace a node labeled e with a new node labeled $\mathbf{let}\ z{=}e\ \mathbf{in}\ z$ which, by driving, would spawn a child labeled e. Thus, no progress has been made.

In such cases a split step is performed. The idea behind a split step is that if the ancestor expression is embedded in the leaf expression, then there is a subterm of the leaf expression which has structure in common with the ancestor. Hence, the split step digs out this structure. For example, in he above case, the leaf expression would be replaced by something similar to $\mathbf{let}\ z{=}\lambda x.e_0\ \mathbf{in}\ z\ e_1$. The parts of the let-expression, which will now be transformed independently from each other, are smaller than the original expression, and this is an important part of the termination proof.

In conclusion, the generalization operations used in positive supercompilation are defined in Figure 1.

Note that the abstract operation is defined only in case $t(\alpha), t(\beta)$ are not let-expressions. This is fortunate since $\sqcap$ is defined only on such expressions. But will we not need to invoke the operations in cases where $t(\alpha), t(\beta)$ are let-expressions? No: let-expressions will be driven without comparison with ancestors.

4.4 Positive supercompilation

We are finally ready to define our first variant of positive supercompilation.

Definition 4.7.

1. An expression is called *terminal* if it is a variable or a 0-ary constructor.
2. An expression is called *trivial* if it is an n-ary constructor ($n > 0$) or an abstraction or of form $R[x]$ where R is not empty, or of form $R[\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]$.

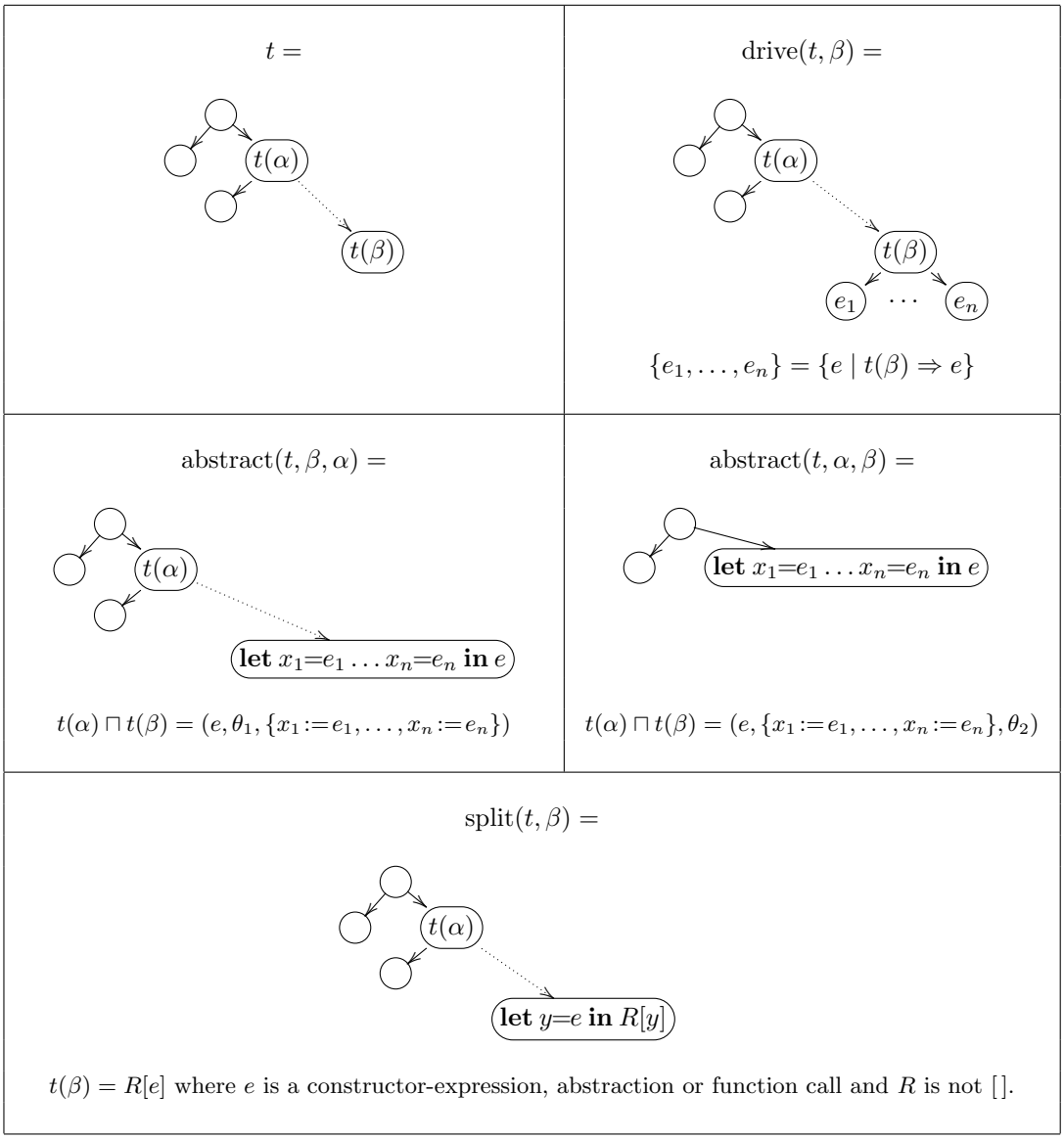


Figure 1: Operations used in Positive Supercompilation

3. For a node α in a tree t, we denote by $t(\alpha)$ the expression in node α.
4. The relevant ancestors of a node β in tree t, denoted $\text{relanc}(t, \beta)$, is the empty set if $t(\beta)$ is trivial and otherwise all non-trivial ancestors of β in t.
5. A leaf β is *processed* in tree t, if if $t(\beta)$ is terminal or $t(\beta)$ is a renaming of $t(\alpha)$ for a relevant ancestor α (in the latter case, folding will be performed).
6. The tree t is *closed* if all leafs in t are processed.

Positive supercompilation can then be defined as follows.[3]

[3]A number of choices are left open in the algorithm, e.g. how one chooses among the unprocessed leaf nodes.

Definition 4.8. Given tree t, if t is closed $P(t) = t$. Otherwise, let β be an unprocessed leaf node and proceed as follows.

if $\forall \alpha \in \text{relanc}(t, \beta) : t(\alpha) \not\trianglelefteq t(\beta)$ **then** $P(t) = \text{drive}(t, \beta)$

else begin

let $\alpha \in \text{relanc}(t, \beta)$ and $t(\alpha) \trianglelefteq t(\beta)$.
if $t(\alpha) \leq t(\beta)$ **then** $P(t) = \text{abstract}(t, \beta, \alpha)$
else if $t(\alpha) \leftrightarrow t(\beta)$ **then** $P(t) = \text{split}(t, \beta)$
else $P(t) = \text{abstract}(t, \alpha, \beta)$.

end

The algorithm calls abstract and split only in cases where these operations are well-defined. Indeed, when $\text{abstract}(t, \beta, \alpha)$ is called, then $\alpha \in \text{relanc}(t, \beta)$. In particular, α, β are non-trivial, so $t(\alpha), t(\beta)$ are not let-expressions. Similarly, when $\text{abstract}(t, \alpha, \beta)$ is called. Finally, when $\text{split}(t, \beta)$ is called, then β is non-trivial.

Proposition 4.9. *P terminates.*

Proof. Similar to the proof for the first-order case in [15]. To see that there are no infinite sequences of direct driving steps, notice that each step decreases the lexicographically ordered measure (m, n) of an expression e, where m is the total number of case-expressions and applications in e, and n is the size of e. □

4.5 Local driving

Note that if we apply the algorithm of the preceding subsection to the example in Section 3 we have several embeddings, starting already when the first child is added under the root. In fact, whenever we unfold a recursive definition, the resulting node will have its predecessor embedded. It is therefore necessary with some form of policy permitting us to drive a non-trivial expression even in the case where there is embedding. What we want is to only check for embedding when encountering terms of form $R[f]$ and only compare to ancestors of the same shape $R'[f']$. This will not only permit us to continue driving but effectively ignore other expressions than those immediately prior to function unfoldings.

In other words, our aim is to extend the notion of trivial node to all nodes except those of form $R[f]$. However, the resulting algorithm does not necessarily terminate, e.g. when applied to Ω. We now devise the solution to that problem.

Definition 4.10 (Duplicator)**.**

1. An abstraction $\lambda x\ .e$ is a *duplicator* if x occurs more than once in e.
2. A case-expression **case** e_0 **of** $p_1 \to e_1;\ \cdots p_n \to e_n;$ is a duplicator (at p_i) if $p_i = c\ x_1 \ldots x_m$ where x_j occurs more than once in e_i, for some i and j.
3. An expression that contains a duplicator is itself a duplicator.

Definition 4.11 (Self-duplicator)**.**

1. An expression $R[(\lambda x.e_0)\ e_1]$ is a *self-duplicator* if both $\lambda x.e_0$ and e_1 are duplicators.
2. $R[$**case** $(c\ e_1 \ldots e_n)$ **of** $\ldots (c\ x_1 \ldots x_n) : e; \ldots]$ is a self-duplicator if the case expression is a duplicator at $c\ x_1 \ldots x_n$ and at least one e_i is a duplicator.

Such details are beyond the scope of the present paper.

Example 4.12. As the following examples illustrate, the informal notion of "having Ω embedded" has been made precise in terms of "being a self-duplictor."

1. The simplest duplicator is $\omega = \lambda x.x\ x$ and the simplest self-duplicator is $\Omega = \omega\ \omega$. Any self-duplicator will have two disjoint subexpressions that are duplicators, and these occur in the position of a reduction context, where transformation will take place.
2. Another duplicator is $w' = \lambda y.\mathbf{case}\ (F\ y)\ \mathbf{of}\ F\ g \rightarrow g\ g$, and two more self-duplicators are $\omega'\omega$ and $\omega'\ \omega'$. Note that when transforming $\omega'\ \omega'$ we never encounter any duplication abstractions, only case-expressions.

The algorithm from the preceding subsection can now be modified as follows.

Definition 4.13 (Higher-order positive supercompilation with local driving)**.** The definition of *trivial* expression is extended to include expressions of form $R[(\lambda x.e_0)\ e_1]$ and $R[\mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ \ldots (c\ x_1 \ldots x_n) : e; \ldots]$. The definition of P is modified as follows.

if $\forall \alpha \in \text{relanc}(t, \beta) : t(\alpha) \not\trianglelefteq t(\beta)$ **then begin**
  if $t(\beta)$ is a self-duplicator **then** $P'(t) = \text{split}(t, \beta)$
  else $P'(t) = \text{drive}(t, \beta)$
end else begin
  let $\alpha \in \text{relanc}(t, \beta)$ and $t(\alpha) \trianglelefteq t(\beta)$
  if $t(\alpha) \leq t(\beta)$ **then** $P'(t) = \text{abstract}(t, \beta, \alpha)$
  else if $t(\alpha) \leftrightarrow t(\beta)$ **then** $P'(t) = \text{split}(t, \beta)$
  else $P'(t) = \text{abstract}(t, \alpha, \beta)$.
end

Proposition 4.14. *P' terminates.*

Proof. The main point is that any sequence of driving steps decreases the lexigographically ordered measure (m, n, k) on an expression e, where m is the number of duplicators in e, n is the number of abstractions and case-expressions in e, and k is the size of the expression. This is similar to the proof in [15]. □

Note that *no* expressions are memoised other than those of form $R[f]$. The algorithm produces exactly the tree in Section 3.

5 Concluding Remarks

In this paper, we have shown how to ensure the termination of higher-order positive supercompilation without memoising other expressions than function calls.

In the higher-order formulations of positive supercompilation given by Mitchell [13] and Bolingbroke [2], *all* expressions are memoised. The extra work required for the additional computationally expensive homeomorphic embedding checks is very time consuming, which has been borne out by experimental results. Also, memoising additional expressions to ensure termination may result in a loss of program efficiency since new function calls may be introduced without being offset by corresponding function unfoldings. It should also be pointed out that the implementation of positive supercompilation in [13] will not terminate on programs such as the second one in Section 1. This is because the simplification rules that are applied to terms prior to transformation by positive supercompilation will not terminate for such programs which

use *contravariant* (*negative*) data types. It is argued in [13] that this problem only occurs for contrived programs, and it is also a problem for GHC, which will not terminate when compiling this example program. However, this seems unsatisfactory. It is noted in [13] that this non-termination problem could be avoided by not performing simplification on negative data types. A similar approach was also adopted by Jonsson [6] and Mendel-Gleason [12] by requiring that all types in the input program are *positive*. This also seems unsatisfactory since there are many programs containing negatives types for which transformation will still terminate. Also, such typing schemes are not used in mainstream functional languages.

Rather than memoising all expressions, the approach taken in the higher-order supercompiler HOSC [7, 8, 9] is to restrict this to only those expressions which are considered to be *non-trivial*. In HOSC 1.0 [7], an expression is considered to be non-trivial if it either has a function in the redex or an irreducible expression in the selector of a **case** expression. However, it was subsequently discovered [8] that this was not sufficient to ensure the termination of the supercompiler, because it will not terminate for programs which encode recursion using a data type such as the second example in Section 1. In HOSC 1.1 [8], an expression for which the next transformation step involves a substitution (corresponding to the left hand side of our rules (2) and (3)) is considered to be non-trivial if it satisfies a size constraint in which the expression resulting from the substitution is no smaller than the expression before substitution. However, it was subsequently discovered [9] that memoising every expression for which the next transformation step involves a β-reduction produces poor residual programs. In HOSC 1.5 [9], expressions for which the next transformation step involves a β-reduction are not memoised, but all applications and **case** expressions are, thus ensuring that in any potentially infinite sequence of transformation steps expressions will still be memoised. However, this may result in a loss of program efficiency since new function calls may be introduced without being offset by corresponding function unfoldings.

In previous work by Hamilton [5], a simple pre-processing step is applied to higher-order programs prior to transformation by positive supercompilation to ensure that in any potentially infinite sequence of transformation steps there must be an unfolding. This involves introducing names for some anonymous functions to ensure that only memoising expressions immediately preceding an unfold step is sufficient to ensure termination of the transformation. Again, this may result in a loss of program efficiency since new function calls may be introduced without being offset by the unfolding of calls of functions from the original program.

The formulation of higher-order positive supercompilation given in this paper goes beyond these previous lines of work by memoising fewer expressions resulting in less loss of program and transformation efficiency, while still ensuring termination.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

References

[1] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1–2):25–46, 1993.

[2] M. Bolingbroke and S. Peyton Jones. Supercompilation by Evaluation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, pages 135–146 , 2010.

[3] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier, 1990.

[4] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. In W. Faber and N. Leone, editors, *25th GULP annual conference*, volume 13 of *Theory and Practice of Logic Programming*, pages 175–199. Cambridge University Press, March 2013.

[5] G.W. Hamilton. On the Termination of Higher-Order Positive Supercompilation. In *Proc. of the First International Workshop on Verification and Program Transformation*, pages 42–56, 2013.

[6] P. Jonsson. *Time- and Size-Efficient Supercompilation.* PhD thesis, Dept. of Computer Science and Electrical Engineering, Lulea University of Technology, 2011.

[7] I. Klyuchnikov. Supercompiler HOSC 1.0: Under the Hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.

[8] I. Klyuchnikov. Supercompiler HOSC 1.1: Proof of Termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[9] I. Klyuchnikov. Supercompiler HOSC 1.5: Homeomorphic Embedding and Generalization in a Higher-Order Setting. Preprint 62, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[10] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of the International Static Analysis Symposium, Pisa, Italy*, pages 230–245, 1998.

[11] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle.* PhD thesis, Université de Nice - Sophia Antipolis, 1994.

[12] G. Mendel-Gleason. *Types and Verification for Infinite State Systems.* PhD thesis, School of Computing, Dublin City University, 2012.

[13] N. Mitchell. Rethinking Supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 309–320 , 2010.

[14] M.H. Sørensen. Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[15] M.H. Sørensen. Convergence of Program Transformers in the Metric Space of Trees. *Lecture Notes in Computer Science*, 1422:315–337, 1998.

[16] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *Lecture Notes in Computer Science*, 787:335–351, 1994.

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

[18] V.F. Turchin. Program Transformation by Supercompilation. *Lecture Notes in Computer Science*, 217:257–281, 1985.

[19] V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.

[20] F. van Raamsdonk, P. Severi, M.H. Sørensen, and H. Xi. Perpetual Reductions in Lambda-Calculus. *Information and Computation*, 149(2):173–225, March 1999.