Introduction to Supercompilation

Morten Heine B. Sørensen and Robert Glück

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
rambo@diku.dk, glueck@diku.dk

Abstract. This paper gives an introduction to Turchin's *supercompiler*, a program transformer for functional programs which performs optimizations beyond *partial evaluation* and *deforestation*. More precisely, the paper presents *positive supercompilation*.

1 Introduction

The *Refal project*—or the *supercompiler project* as we shall call it—was conceived in the mid 1960's by Valentin F. Turchin and co-workers in Russia.

A central idea in the supercompiler project is that of a *metasystem transition* [25]: a jump from a system S to a metasystem S' that integrates, modifies, and controls a number of S-systems as its subsystems. Turchin considers metasystem transition as a key to creative, human thinking—others are *computation* (or *deduction*) and *generalization* (or *abstraction* or *induction*).

As an illustration of these ideas, consider how we might solve some scientific problem. We observe phenomena, generalize observations, and try to construct a self-sufficient model of the reality we observe in terms of these generalizations. If we fail to obtain a solution, we start to analyze why we failed, and for this purpose we examine the process of applying our methods for solving the problem; we perform a metasystem transition with respect to the ground-level set of rules. This may give us new, more elaborate methods to solve the problem. If we fail once more, we make another metasystem transition and analyze our means of finding new rules. These transitions may proceed infinitely.

The supercompiler project [26, 27, 29] is a product of this cybernetic thinking. A program is seen as a machine and to make sense of it, one must control and observe (*super*vise) its operations. A supercompiler is a program transformer that creates a graph of configurations and transitions between possible configurations of the computing system. This process, called *driving*, will usually go on infinitely. To make it finite and self-sufficient (*closed*) a supercompiler performs *generalization* on the systems configurations. Application of the supercompiler to a program can be seen as a metasystem transition, and repeated metasystem transition can be expressed by self-application of the supercompiler.

One of the motivations [27] for the project comes from artificial intelligence. If metasystem transition is taken to be one of the main sources for creative thinking, an implementation on a computer of the concept would indeed seem an interesting approach to artificial intelligence.

Another motivation [26] for the supercompiler project is related to a particular instance of metasystem transition. The appearance of numerous programming languages, some for application specific purposes, motivated the idea of a programming environment that facilitates the introduction of new languages and hierarchies of languages. The language *Refal* [27], developed by Turchin, can be viewed as a meta-algorithmic language: the specification of a new language L is an L-interpreter written in Refal. In such a setting one needs a means of turning programs written in application specific languages into efficient programs, preferably in the ground-level language. This task is undertaken by the supercompiler written for Refal programs, and implemented in Refal—Turchin independently realized all three *Futamura projections* stated in terms of metasystem transition.

A number of other applications of metasystem transition in the computer have emerged, e.g., [7, 11, 17, 28]. The most widely appreciated application of supercompilation is as a *program optimizer* performing *program specialization* and *elimination of intermediate data structures* as well as more dramatic optimizations. This is the only application of supercompilation we shall be concerned with in this paper.

From its very inception, Turchin's supercompiler was formulated for the language Refal. The authors have since reconsidered the theory of supercompilation—the part dealing with program optimizations—in the context of a more familiar functional language [6, 21]. A variant of supercompilation, called *positive supercompilation,* was developed in an attempt to understand the essence of supercompilation, how it achieves its effects, and its correspondence to related transformers [8, 9, 18, 22, 23]. For this variant, results were developed dealing with the problems of preserving semantics, evaluating efficiency of transformed programs, and ensuring termination [21, 24].

Turchin developed the philosophy underlying the supercompiler project in [25]. His recent account [29] contains historical information and an introduction to several aspects of supercompilation—as seen by Turchin. Experiments with Turchin's supercompilation have been reported [30, 27, 17], see also [29]. Other experimental systems for supercompilation have been developed, e.g. [6, 31, 13], including earlier systems by the Refal group (most unpublished). The first non-Refal supercompiler was [6]. A comprehensive bibliography can be found in [9].

This paper presents positive supercompilation following [6, 8, 9, 21–24]. We first present examples of driving and generalization (Section 2). After some preliminaries (Section 3) we introduce driving (Section 4), illustrated by a classical application (Section 5). We then present generalization (Section 6), which can be viewed as a technique to ensure termination of driving. After some more examples (Section 7), the paper ends with some remarks about correctness and the relation to other program transformers (Section 8).

2 Examples of Driving and Generalization

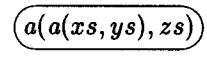
This section illustrates driving and generalization by means of examples. The first example primarily illustrates driving, the second example generalization.

Example 1. Consider a functional program appending two lists.

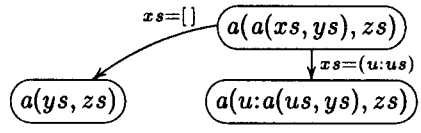
$$\begin{aligned} a([], vs) &= vs \\ a(u{:}us, vs) &= u{:}a(us, vs) \end{aligned}$$

A simple and elegant way to append *three* lists is to use the expression $a(a(xs, ys), zs)$. However, this expression is inefficient since it traverses xs twice. We now illustrate a standard transformation obtaining a more efficient method.

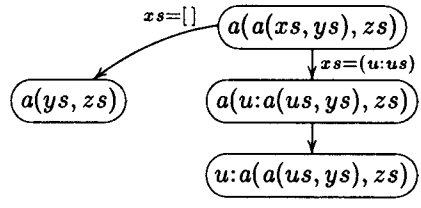
We begin with a tree whose single node is labeled with $a(a(xs, ys), zs)$:



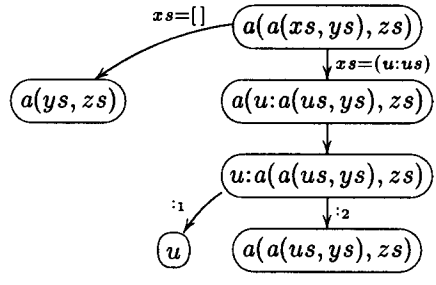
Whenever the reduction step has different possible outcomes, new children are added to account for all possibilities. By a *driving step* which replaces the inner call to append according to the different patterns in the definition of a, two new expressions are added as labels on children:



In the rightmost child we can perform a driving step which replaces the outer call to append:

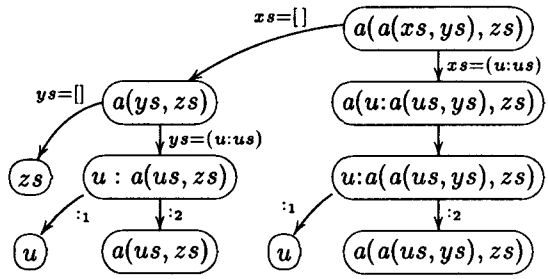


The label of the new child contains an outermost constructor. For transformation to propagate to the subexpression of the constructor we again add children:



The expression in the rightmost child is a renaming of the expression in the root; that is, the two expressions are identical up to choice of variable names. As we shall see below, no further processing of such a node is required. Driving

the child with label $a(ys, zs)$ two steps leads to:



The tree is now *closed* in the sense that each leaf expression either is a renaming of an ancestor's expression, or contains a variable or a 0-ary constructor. Informally, a closed tree is a representation of all possible computations with the expression e in the root, where branchings in the tree correspond to different run-time values for the variables of e. The nodes can be perceived as states, and the edges correspond to transitions.

In the above tree, computation starts in the root, and then branches to one of the successor states depending on the shape of xs. Assuming xs has form $(u{:}us)$, the constructor ":" is then emitted and control is passed to the two states corresponding to nodes labeled u and $a(a(us, ys), zs)$, etc.

To construct a new program from the closed tree, we introduce, roughly, for each node α with child β a definition where the left and right hand side of the definition are derived from α and β, respectively. More specifically, we rename expressions of form $a(a(xs, ys), zs)$ and $a(ys, zs)$ as $aa(xs, ys, zs)$ and $a'(ys, zs)$, respectively, and derive from the tree the following new program:

$$\begin{aligned} aa([\,], ys, zs) &= a'(ys, zs) \\ aa(u{:}us, ys, zs) &= u : aa(us, ys, zs) \end{aligned}$$

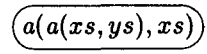
$$\begin{aligned} a'([\,], zs) &= zs \\ a'(u{:}us, zs) &= u{:}a'(us, zs) \end{aligned}$$

The expression $aa(\,xs,\ ys,\ zs)$ in this program is more efficient than the expression $a(a(xs, ys), zs)$ in the original program, since the new expression traverses xs only once.

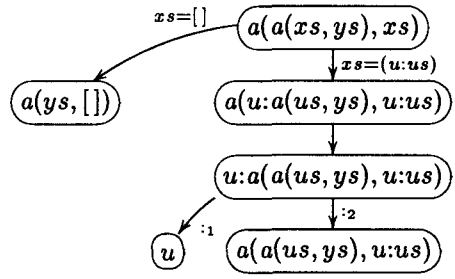
The transformation in Example 1 proceeded in three phases the first two of which were interleaved. In the first phase we performed driving steps that added children to the tree. In the second phase we made sure that no node with an expression which was a renaming of an ancestor's expression was driven, and we continued the overall process until the tree was closed. In the third phase we recovered from the resulting finite, closed tree a new expression and program.

In the above transformation we ended up with a finite closed tree. Often, special measures must be taken to ensure that this situation is eventually encountered, as in the next example.

Example 2. Suppose we want to transform the expression $a(a(xs, ys), xs)$, where a is defined as in Example 1—note the double occurrence of xs. As above we start out with:

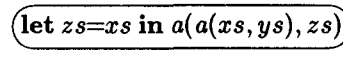


After the first few steps we have:

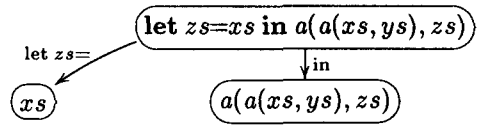


Unlike the situation in Example 1, the label of the rightmost node is not a renaming of the expression at the root. In fact, repeated driving will *never* lead to that situation; special measures must be taken.

One solution is to ignore the information that the argument xs to the inner call and the argument xs to the outer call are the same. This is achieved by a *generalization step* that replaces the whole tree by a single new node:



When dealing with nodes of the new form **let** $zs{=}e$ **in** e' we then transform e and e' independently. Thus we arrive at:



Driving of the node labeled $a(a(xs, ys), zs)$ leads to the last tree in Example 1.

When generating a new term and program from such a tree, we can eliminate all let-expressions; in particular, in the above example, we generate the expression $aa(xs, ys, xs)$ and the same program as in Example 1. In some cases such let-expression elimination may be undesirable for reasons pertaining to efficiency of the generated program—but such issues are ignored in the present paper.

Again transformation proceeds in three phases, but the second phase is now more sophisticated, sometimes replacing a subtree by a new node in a generalization step.

The positive supercompiler is an algorithm that repeatedly applies driving and generalization steps until a closed tree is encountered from which a new program can then be recovered. In Section 4 we rigorously define driving steps and in Section 6 we similarly define generalization steps along with the positive supercompilation algorithm.

3 Preliminaries

This section introduces preliminaries on trees, programs, and substitutions.

3.1 Trees

We introduce trees in a rigorous manner, following Courcelle [1].

Definition 1. A *tree* over a set E is a partial map[1] $t : \mathbf{N}_1^* \to E$ such that

1. $\mathrm{dom}(t) \neq \emptyset$ (t is *non-empty*);
2. if $\alpha\beta \in \mathrm{dom}(t)$ then $\alpha \in \mathrm{dom}(t)$ ($\mathrm{dom}(t)$ is *prefix-closed*);
3. if $\alpha \in \mathrm{dom}(t)$ then $\{i \mid \alpha i \in \mathrm{dom}(t)\}$ is finite (t is *finitely branching*);
4. if $\alpha j \in \mathrm{dom}(t)$ then $\alpha i \in \mathrm{dom}(t)$ for all $1 \leq i \leq j$ (t is *ordered*).

Let t be a tree over E. The elements of $\mathrm{dom}(t)$ are called *nodes* of t; the empty string ϵ is the *root*, and for any node α in t, the nodes αi of t (if any) are the *children* of α, and we also say that α is the *parent* of these nodes. A node α in t is *to the left* of another node β in t if there is a node γ in t such that $\alpha = \gamma i \gamma'$ and $\beta = \gamma j \gamma''$, where $i < j$. A *branch* in t is a finite or infinite sequence $\alpha_0, \alpha_1, \ldots \in \mathrm{dom}(t)$ where, for all i, α_{i+1} is a child of α_i. A node with no children is a *leaf*. We denote by $\mathrm{leaf}(t)$ the set of all leaves in t. For any node α of t, $t(\alpha) \in E$ is the *label* of α. Also, t is *finite*, if $\mathrm{dom}(t)$ is finite. Finally, t is *singleton* if $\mathrm{dom}(t) = \{\epsilon\}$, i.e., if $\mathrm{dom}(t)$ is singleton. $T(E)$ is the set of all finite trees over E.

Example 3. Let $\mathcal{E}_H(V)$ be the set of expressions over symbols H and variables V. Let $x, xs, \ldots \in V$ and $a, cons, nil \in H$, denoting $(x{:}xs)$ by $cons(x, xs)$ and $[]$ by nil. The trees in Example 1 (ignoring labels on edges) are a diagrammatic presentation of trees over $\mathcal{E}_H(V)$.

The following notions pertaining to trees will be used in the remainder.

Definition 2. Let E be a set, and $t, t' \in T(E)$.

1. For $\alpha \in \mathrm{dom}(t)$, $t\{\alpha := t'\}$ denotes the tree t'' defined by:

$$\mathrm{dom}(t'') = (\mathrm{dom}(t) \setminus \{\alpha\beta \mid \alpha\beta \in \mathrm{dom}(t)\}) \cup \{\alpha\beta \mid \beta \in \mathrm{dom}(t')\}$$
$$t''(\gamma) = \begin{cases} t'(\beta) \text{ if } \gamma = \alpha\beta \text{ for some } \beta \\ t(\gamma) \text{ otherwise} \end{cases}$$

2. We write $t = t'$, if $\mathrm{dom}(t) = \mathrm{dom}(t')$ and $t(\alpha) = t'(\alpha)$ for all $\alpha \in \mathrm{dom}(t)$.
3. Let $\alpha \in \mathrm{dom}(t)$. The *ancestors of* α *in* t is the set

$$\mathrm{anc}(t, \alpha) = \{\beta \in \mathrm{dom}(t) \mid \exists \gamma : \alpha = \beta\gamma\}$$

[1] We let $\mathbf{N}_1 = \mathbf{N} \setminus \{0\}$. S^* is the set of finite strings over S, and $\mathrm{dom}(f)$ is the domain of a partial function f. We use α, β to denote elements of $\mathbf{N}_1^*$ and i, j to denote elements of $\mathbf{N}_1$.

4. We denote by $e \rightarrow e_1, \ldots, e_n$ the tree $t \in T(E)$ with

$$\begin{aligned} \mathrm{dom}(t) &= \{\epsilon\} \cup \{1, \ldots, n\} \\ t(\epsilon) &= e \\ t(i) &= e_i \end{aligned}$$

As a special case, $e \rightarrow$ denotes the $t \in T(E)$ with $\mathrm{dom}(t) = \{\epsilon\}$ and $t(\epsilon) = e$.

The tree $t\{\alpha{:=}t'\}$ is the tree obtained by replacing the subtree with root α in t by the tree t'. The ancestors of a node are the nodes on the path from the root to the node (including the node itself). Finally, the tree $e \rightarrow e_1, \ldots, e_n$ is the tree with root labeled e and n children labeled $e_1, \ldots, e_n$, respectively.

3.2 Programs

We consider the following first-order functional language; the intended operational semantics is normal-order graph reduction to weak head normal form.

Definition 3. We assume a denumerable set of symbols for variables $x \in X$ and finite sets of symbols for constructors $c \in C$, and functions $f \in F$ and $g \in G$, where X, C, F, and G are pairwise disjoint. All symbols have fixed arity. The sets $\mathcal{Q}$ of programs, $\mathcal{D}$ of definitions, $\mathcal{E}$ of expressions, and $\mathcal{P}$ of patterns are defined by:

$$\begin{array}{lcll} \mathcal{Q} \ni q & ::= & d_1 \ldots d_m & \\ \mathcal{D} \ni d & ::= & f(x_1, \ldots, x_n) = e & \text{(f-function)} \\ & | & g(p_1, x_1, \ldots, x_n) = e_1 & \\ & & \vdots & \text{(g-function)} \\ & & g(p_m, x_1, \ldots, x_n) = e_m & \\ \mathcal{E} \ni e & ::= & x & \text{(variable)} \\ & | & c(e_1, \ldots, e_n) & \text{(constructor)} \\ & | & f(e_1, \ldots, e_n) & \text{(f-function call)} \\ & | & g(e_0, e_1, \ldots, e_n) & \text{(g-function call)} \\ & | & \textbf{if } e_1{=}e_2 \textbf{ then } e_3 \textbf{ else } e_4 & \text{(conditional with equality test)} \\ \mathcal{P} \ni p & ::= & c(x_1, \ldots, x_n) & \end{array}$$

where $m > 0, n \geq 0$. We require that no two patterns p_i and p_j in a g-function definition contain the same constructor c, that no variable occur more than once in a left side of a definition, and that all variables in the right side of a definition be present in its left side. By vars(e) we denote the set of variables occurring in the expression e.

Example 4. The append program in Example 1 is a program in this language using the short notation [] and $(x : xs)$ for the list constructors *nil* and $cons(x, xs)$.

Remark 1. In some accounts of positive supercompilation the language contains case-expressions instead of g-functions (pattern-matching definitions). The difference between *deforestation* [32] and positive supercompilation is clearest in presentations with case-expressions, but the formulation of generalization steps is simplest in presentations with pattern-matching definitions.

Remark 2. There is a close relationship between the set $\mathcal{E}$ of expressions introduced in Definition 3 and the set $\mathcal{E}_H(V)$ introduced in Example 3. In fact, $\mathcal{E} = \mathcal{E}_{C \cup F \cup G \cup \{\mathbf{if}\}}(X)$, where we view **if** as a 4-ary operator symbol. Therefore, we can make use of well-known facts about $\mathcal{E}_H(V)$ in reasoning about $\mathcal{E}$.

3.3 Substitutions

Finally we introduce substitutions and some related operations.

Definition 4.

1. A *substitution* on $\mathcal{E}_H(V)$ is a total map from V to $\mathcal{E}_H(V)$. Substitutions are lifted to expressions as usual. Application of substitutions is written postfix.
2. A substitution θ is a *renaming* if it is a bijection from V to $V \subseteq \mathcal{E}_H(V)$.
3. If θ is a renaming and $\theta_2 = \theta \circ \theta_1$ then θ_2 is a renaming of θ_1; if $e_2 = e_1\theta$ then e_2 is a renaming of e_1.
4. We denote by $\{x_1{:=}e_1, \ldots, x_n{:=}e_n\}$ the substitution that maps x_i to e_i and all other variables to themselves.

Definition 5.

1. Two expressions e_1, e_2 are *unifiable* if there exists a substitution θ such that $e_1\theta = e_2\theta$. Such a θ is a *unifier*. Moreover, θ is a *most general unifier* (mgu) if, for any other unifier θ', there is a substitution σ such that $\theta' = \sigma \circ \theta$.
2. Two expressions e_1, e_2 are *disunifiable* if there exists a substitution θ such that $e_1\theta \neq e_2\theta$.

Proposition 1. *Let H, V be some sets. If $e_1, e_2 \in \mathcal{E}_H(V)$ are unifiable, then they have exactly one mgu modulo renaming (of substitutions).*

Proof. See [14]. □

Definition 6. By $e_1 \sqcup e_2$ we denote some mgu of e_1 and e_2.

Definition 7. Let $e_1, e_2 \in \mathcal{E}_H(V)$, for some H, V.

1. The expression e_2 is an *instance* of e_1, $e_1 \lesssim e_2$, if $e_1\theta = e_2$ for a substitution θ.
2. A *generalization* of e_1, e_2 is an expression e_g such that $e_g \lesssim e_1$ and $e_g \lesssim e_2$.
3. A *most specific* generalization (msg) of e_1 and e_2 is a generalization e_g such that, for every generalization e'_g of e_1 and e_2, it holds that $e'_g \lesssim e_g$.

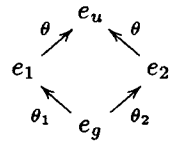
Proposition 2. *Let H, V be some sets. Any two $e_1, e_2 \in \mathcal{E}_H(V)$ have exactly one msg modulo renaming (of expressions).*

Proof. See [14]. □

Definition 8. By $e_1 \sqcap e_2$ we denote some msg of e_1 and e_2.[2] Two expressions e_1 and e_2 are *incommensurable*, $e_1 \leftrightarrow e_2$, if $e_1 \sqcap e_2$ is a variable.

[2] As a matter of technicality, we shall require that if $e_1 \lesssim e_2$ then $e_1 \sqcap e_2 = e_1$. In other words, whenever e_2 is an instance of e_1, the variable names of $e_1 \sqcap e_2$ will be chosen so that the resulting term is identical to e_1.

Generalization can be viewed as a dual to unification. Whereas a unifier of e_1 and e_2 is a *substitution* θ that maps both e_1 and e_2 to some e_u, a generalization of e_1 and e_2 is an *expression* e_g such that both e_1 and e_2 are instances of e_g:



Remark 3. Note that we now use the term *generalization* in two distinct senses: to denote certain operations on trees performed by supercompilation (cf. Example 2), and to denote the above operation on expressions. The two senses are related: generalization in the former sense will make use of generalization in the latter sense.

Example 5. The following illustrates most specific generalizations $e_1 \sqcap e_2$ of $e_1, e_2 \in \mathcal{E}_H(V)$ where $(e_1 \sqcap e_2)\theta_i = e_i$ and $x, y \in V$, $b, c, d, f \in H$.

e_1	e_2	$e_1 \sqcap e_2$	θ_1	θ_2
b	$f(b)$	x	$\{x := b\}$	$\{x := f(b)\}$
$c(b)$	$c(f(b))$	$c(x)$	$\{x := b\}$	$\{x := f(b)\}$
$c(y)$	$c(f(y))$	$c(y)$	$\{\}$	$\{y := f(y)\}$
$d(b,b)$	$d(f(b),f(b))$	$d(x,x)$	$\{x := b\}$	$\{x := f(b)\}$

Remark 4. An msg of $e, e' \in \mathcal{E}_H(V)$ and the corresponding substitutions can be obtained by exhaustively applying the following rewrite rules to the initial triple $(x, \{x := e\}, \{x := e'\})$:

$$\begin{pmatrix} e_g \\ \{x := h(e_1, \ldots, e_n)\} \cup \theta_1 \\ \{x := h(e'_1, \ldots, e'_n)\} \cup \theta_2 \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} e_g\{x := h(y_1, \ldots, y_n)\} \\ \{y_1 := e_1, \ldots, y_n := e_n\} \cup \theta_1 \\ \{y_1 := e'_1, \ldots, y_n := e'_n\} \cup \theta_2 \end{pmatrix}$$

$$\begin{pmatrix} e_g \\ \{x := e, y := e\} \cup \theta_1 \\ \{x := e', y := e'\} \cup \theta_2 \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} e_g\{x := y\} \\ \{y := e\} \cup \theta_1 \\ \{y := e'\} \cup \theta_2 \end{pmatrix}$$

4 Driving

We now present the driving steps mentioned in Section 2, as used in positive supercompilation.

4.1 Driving Step

When we perform driving steps, we instantiate variables to patterns, e.g., *xs* to (*u*:*us*). To avoid confusion of variables, we must choose the variables in the pattern with some care. The following definition introduces useful terminology to deal with that problem.

Definition 9. A substitution θ is *free for* an expression $e \in \mathcal{E}_H(V)$ if for all $x \in V$: $(\mathrm{vars}(x\theta)\backslash\{x\}) \cap \mathrm{vars}(e) = \emptyset$.

The crucial property of a substitution θ which is free for an expression e is, roughly, that the variables in the range of θ (at least those variables that are not simply mapped to themselves) do not occur already in e.

When we drive conditionals we distinguish the case where no transformations are possible for the two expressions in the equality test. The following class of terms is useful for that purpose.

Definition 10. The sets $\mathcal{B}$ of basic values where $n \geq 0$, is defined by:

$$\mathcal{B} \ni b \quad ::= \quad x \mid c(b_1, \ldots, b_n)$$

The following relation $\Rightarrow$ is similar to the small-step semantics for normal-order reduction to weak head normal form (the outermost reducible subexpression is reduced). However, $\Rightarrow$ also propagates to the arguments of constructors and works on expressions with variables; the latter is done by *unification-based information propagation* [6, 9] (the assumed outcome of an equality test and constructor test is propagated by a substitution—notice, e.g., the substitution $\{y := p\}$ in the third rule).

Definition 11. For a program q, the relations $e \Rightarrow e'$, $e \rightarrow_\theta e'$, and $e \twoheadrightarrow_\theta e'$ where $e, e' \in \mathcal{E}$ and θ is a substitution on $\mathcal{E}$, are defined in Figure 1.

Example 6. The rules (1)-(5) are the base cases. For instance,

$$a(xs, ys) \rightarrow_{\{xs:=(u:us)\}} u : a(us, ys) \qquad \text{Rule (3)}$$

The rules (6), (9), and (10) allow reduction in contexts, i.e., inside the first argument of a g-function and inside the test in a conditional. For instance,

$$a(a(xs, ys), xs) \rightarrow_{\{xs:=(u:us)\}} a(u : a(us, ys), xs) \qquad \text{Rule (6)}$$

The rules (9) and (10) are more complicated than (6). The reason is that in the former rules we are allowed to reduce under constructors, because the expressions in the test must be reduced to basic values; for the first argument in a g-function we only need an expression with an outermost constructor.

Finally, the rules (11) and (12) are the main rules. For instance,

$$\begin{array}{lll} a(a(xs, ys), xs) & \Rightarrow a(u : a(us, ys), u : us) & \text{Rule (11)} \\ u : a(a(us, ys), u : us) & \Rightarrow a(a(us, ys), u : us) & \text{Rule (12)} \end{array}$$

Remark 5. In some accounts of positive supercompilation we use so-called *contexts* and *redexes* instead of the above inference rules to define the relation $\Rightarrow$. The choice between contexts and redexes on the one hand and inference rules on the other hand is mostly a matter of taste.

Definition 12. Let $t \in T(\mathcal{E})$ and $\beta \in \mathrm{leaf}(t)$. Then

$$\mathrm{drive}(t, \beta) = t\{\beta := t(\beta) \rightarrow e_1, \ldots, e_n\}$$

where $\{e_1, \ldots, e_n\} = \{e \mid t(\beta) \Rightarrow e\}$.

Example 7. All the steps in Example 1 are, in fact, driving steps in this sense.

Base Cases:

$$\frac{f(x_1,\ldots,x_n) = e \ \in \ q}{f(e_1,\ldots,e_n) \rightarrow_{\{\}} e\{x_1 := e_1,\ldots,x_n := e_n\}} \quad (1)$$

$$\frac{g(c(x_1,\ldots,x_m),x_{m+1},\ldots,x_n) = e \ \in \ q}{g(c(e_1,\ldots,e_m),e_{m+1},\ldots,e_n) \rightarrow_{\{\}} e\{x_1 := e_1,\ldots,x_n := e_n\}} \quad (2)$$

$$\frac{g(p,x_1\ldots,x_n) = e \ \in \ q}{g(y,e_1,\ldots e_n) \rightarrow_{\{y:=p\}} e\{x_1 := e_1,\ldots,x_n := e_n\}} \quad (3)$$

$$\frac{b_1 \text{ unifiable with } b_2}{\textbf{if } b_1{=}b_2 \textbf{ then } e_3 \textbf{ else } e_4 \rightarrow_{b_1 \sqcup b_2} e_3} \quad (4)$$

$$\frac{b_1 \text{ disunifiable with } b_2}{\textbf{if } b_1{=}b_2 \textbf{ then } e_3 \textbf{ else } e_4 \rightarrow_{\{\}} e_4} \quad (5)$$

Reduction in Context:

$$\frac{e \rightarrow_\theta e'}{g(e,e_1,\ldots,e_n) \rightarrow_\theta g(e',e_1,\ldots,e_n)} \quad (6)$$

$$\frac{e \rightarrow_\theta e'}{e \twoheadrightarrow_\theta e'} \quad (7)$$

$$\frac{e \twoheadrightarrow_\theta e'}{c(b_1,\ldots,b_i,e,e_{i+1},\ldots,e_n) \twoheadrightarrow_\theta c(b_1,\ldots,b_i,e',e_{i+1},\ldots,e_n)} \quad (8)$$

$$\frac{e_1 \twoheadrightarrow_\theta e_1'}{\textbf{if } e_1{=}e_2 \textbf{ then } e_3 \textbf{ else } e_4 \rightarrow_\theta \textbf{if } e_1'{=}e_2 \textbf{ then } e_3 \textbf{ else } e_4} \quad (9)$$

$$\frac{e_2 \twoheadrightarrow_\theta e_2'}{\textbf{if } b_1{=}e_2 \textbf{ then } e_3 \textbf{ else } e_4 \rightarrow_\theta \textbf{if } b_1{=}e_2' \textbf{ then } e_3 \textbf{ else } e_4} \quad (10)$$

Driving Step:

$$\frac{e \rightarrow_\theta e' \ \ \& \ \ \theta \text{ is free for } e'}{e \Rightarrow e'\theta} \quad (11)$$

$$\frac{i \in \{1,\ldots,n\}}{c(e_1,\ldots,e_n) \Rightarrow e_i} \quad (12)$$

Fig. 1. Driving step

4.2 Driving with Folding

The positive supercompiler is an algorithm which applies driving steps interleaved with generalization steps in the style of Example 2. It will be considered in Section 6. However, the driving step is powerful enough on its own to achieve fascinating effects. We therefore consider next an algorithm which only performs driving steps and looks for recurring nodes as in Example 1. This strategy is known as *α-identical folding* [23] (the same approach is taken in deforestation [32]). The following algorithm will be called *driving with identical folding.*

Definition 13. Let $t \in T(\mathcal{E})$. A $\beta \in \text{leaf}(t)$ is *processed* if one of the following conditions are satisfied:

1. $t(\beta) = c()$ for some $c \in C$;
2. $t(\beta) = x$ for some $x \in X$;
3. there is an $\alpha \in \text{anc}(t, \beta) \backslash \{\beta\}$ such that $t(\alpha)$ is a renaming of $t(\beta)$.

Also, t is *closed* if all leaves in t are processed.

The driving algorithm can then be defined as follows.[3]

Definition 14. Let q be a program, and define $M_d : \mathcal{E} \to T(\mathcal{E})$ by:

input $e \in \mathcal{E}$;
let $t = e\!\rightarrow$;
while t is not closed
  let $\beta \in \text{leaf}(t)$ be an unprocessed node;
  let $t = \text{drive}(t, \beta)$;
return t;

Example 8. The sequence of steps in Example 1 could be computed by M_d.

5 The Knuth-Morris-Pratt Example

In this section we review one of the classical examples of supercompilation: to generate from a general pattern matcher and a fixed pattern, a specialized pattern matcher of efficiency similar to the one generated by the Knuth-Morris-Pratt algorithm [12]. We will do this with the driving algorithm of Definition 13.

Consider the following *general matcher* which takes a pattern and a string as input and returns *True* iff the pattern occurs as a substring in the string.[4]

$$
\begin{array}{ll}
\mathit{match}(p, s) & = m(p, s, p, s) \\
m([], ss, op, os) & = \mathit{True} \\
m(p{:}pp, ss, op, os) & = x(p, pp, ss, op, os) \\
x(p, pp, [], op, os) & = \mathit{False} \\
x(p, pp, s{:}ss, op, os) & = \textbf{if } p{=}s \textbf{ then } m(pp, ss, op, os) \textbf{ else } n(op, os) \\
n(op, []) & = \mathit{False} \\
n(op, s{:}ss) & = m(op, ss, op, ss)
\end{array}
$$

Now consider the following *naively specialized matcher* $\mathit{match}_{\mathrm{AAB}}$ which matches the fixed pattern $[A,A,B]$ with a string u by calling *match*:

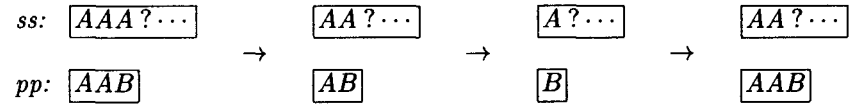
$$\mathit{match}_{\mathrm{AAB}}(u) \quad = \quad \mathit{match}([A,A,B], u)$$

[3] A number of choices are left open in the algorithm, e.g., how one chooses among the unprocessed leaf nodes. Such details are beyond the scope of the present paper. Also, as we shall see, M_d is actually a partial map. Finally, we omit the definition of code generation.

[4] By a small abuse, we permit x to be defined by patterns on the third argument instead of the first argument.

Evaluation proceeds by comparing A to the first component of u, A to the second, B to the third. If at some point the comparison failed, the process is restarted with the tail of u.

This strategy is not optimal. If the string u begins, e.g., with three A's, then the steps of the naively specialized matcher can be depicted as follows:



After matching the two A's in the pattern with the first two A's in the string, the B in the pattern fails to match the A in the string. Then the process is restarted with the string's tail, even though it is known that the first two comparisons will succeed. Rather than performing these tests whose outcome is already known, we should *skip* the three first A's in the original string and proceed directly to compare the B in the pattern with the fourth element of the original string. This is done in the *KMP specialized matcher.*

$$
\begin{array}{ll}
match_{\mathrm{AAB}}(u) & = m_{\mathrm{AAB}}(u) \\
m_{\mathrm{AAB}}([]) & = \mathit{False} \\
m_{\mathrm{AAB}}(s{:}ss) & = \textbf{if } A{=}s \textbf{ then } m_{\mathrm{AB}}(ss) \textbf{ else } m_{\mathrm{AAB}}(ss) \\
m_{\mathrm{AB}}([]) & = \mathit{False} \\
m_{\mathrm{AB}}(s{:}ss) & = \textbf{if } A{=}s \textbf{ then } m_{\mathrm{B}}(ss) \textbf{ else } m_{\mathrm{AAB}}(ss) \\
m_{\mathrm{B}}([]) & = \mathit{False} \\
m_{\mathrm{B}}(s{:}ss) & = \textbf{if } B{=}s \textbf{ then } \mathit{True} \textbf{ else if } A{=}s \textbf{ then } m_{\mathrm{B}}(ss) \textbf{ else } m_{\mathrm{AAB}}(ss)
\end{array}
$$

After finding two A's and a third symbol which is not a B in the string, this program checks (in m_{B}) whether the third symbol of the string is an A. If so, it continues immediately by comparing the next symbol of the string with the B in the pattern (by calling m_{B}), thereby avoiding repeated comparisons.

We will now see how the driving algorithm of the previous section can be used to derive matchers similar to the above one. Figure 2 shows the closed tree that arises by application of M_d to $match([A,A,B],u)$, the body of the naively specialized matcher.

From this tree we can generate the following almost optimal specialized matcher.

$$
\begin{array}{ll}
m_{\mathrm{AAB}}([]) & = \mathit{False} \\
m_{\mathrm{AAB}}(s : ss) & = \textbf{if } A{=}s \textbf{ then } m_{\mathrm{AB}}(ss) \textbf{ else } n_{\mathrm{AAB}}(ss, s) \\
m_{\mathrm{AB}}([]) & = \mathit{False} \\
m_{\mathrm{AB}}(s : ss) & = \textbf{if } A{=}s \textbf{ then } m_{\mathrm{B}}(ss) \textbf{ else } n_{\mathrm{AB}}(ss, s) \\
m_{\mathrm{B}}([]) & = \mathit{False} \\
m_{\mathrm{B}}(s : ss) & = \textbf{if } B{=}s \textbf{ then } \mathit{True} \textbf{ else } n_{\mathrm{B}}(ss, s) \\
n_{\mathrm{AAB}}(ss, s) & = m_{\mathrm{AAB}}(ss) \\
n_{\mathrm{AB}}(ss, s) & = \textbf{if } A{=}s \textbf{ then } m_{\mathrm{AB}}(ss) \textbf{ else } n_{\mathrm{AAB}}(ss, s) \\
n_{\mathrm{B}}(ss, s) & = \textbf{if } A{=}s \textbf{ then } m_{\mathrm{B}}(ss) \textbf{ else } n_{\mathrm{AB}}(ss, s)
\end{array}
$$

The term $m_{\mathrm{AAB}}(u)$ in this program is more efficient than $match([A,A,B],u)$ in the original program. In fact, this is the desired KMP specialized matcher,

disregarding the redundant test $A = s$ in n_{AB}. But these redundant tests do not affect run-time seriously: there is a constant c such that the total number of redundant tests in the entire evaluation of $m_{\mathrm{p}}(ss)$ for any pattern p and subject string ss is bound by $c \cdot |ss|$, as shown in [21].

The reason for the redundant test $A = s$ is that driving—as defined in this paper—ignores *negative* information: when driving proceeds to the false branch of a conditional, the information that the equality does not hold is forgotten. Driving maintains only *positive* information: in the true-branch of a conditional a substitution is performed representing the information that the equality test is assumed to come out true. This explains the terminology *positive* supercompilation! Representing negative information, i.e. the information that an equality does not hold, requires a more sophisticated representation than positive information. A transformer that has the capacity to eliminate all unreachable branches in a program has *perfect* information propagation [6].

6 Generalization

Example 2 showed that the driving algorithm does not always terminate. In this section we add generalization steps in such a way that the resulting positive supercompilation algorithm always terminates.

6.1 Driving Step

As we saw in Example 2, although the input and output programs of the transformer are expressed in the language of Definition 3, the trees considered during transformation might have nodes containing let-expressions. Therefore, the positive supercompiler will work on trees over $\mathcal{L}$, defined as follows.

Definition 15. The set $\mathcal{L}$ of let-expressions is defined by:

$$\mathcal{L} \ni \ell \quad ::= \quad \mathbf{let}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathbf{in}\ e$$

where $n \geq 0$. If $n > 0$ then we say that $\mathcal{L}$ is a *proper* let-expression and require that $x_1, \ldots, x_n \in \mathrm{vars}(e)$, that $e \notin X$, and that $e\{x_1 := e_1, \ldots, x_n := e_n\}$ is not a renaming of e. If $n = 0$ we identify the expression $\mathbf{let}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathbf{in}\ e$ with e. Thus, $\mathcal{E}$ is a subset of $\mathcal{L}$.

Example 9. The trees in Example 2 (ignoring labels on edges) are a diagrammatic presentation of trees over $\mathcal{L}$.

In order to formulate positive supercompilation we must adapt Definition 11 of *driving step* to deal with the new set of expressions. This is done as follows.

Definition 16.

1. For a program q, the relations $\ell \Rightarrow e'$, $e \rightarrow_\theta e'$, and $e \twoheadrightarrow_\theta e'$ where $e, e' \in \mathcal{E}$, $\ell \in \mathcal{L}$, and θ is a substitution on $\mathcal{E}$, are defined as in Figure 1 with the addition of the rule

$$\frac{i \in \{1, \ldots, n+1\}}{\mathbf{let}\ x_1{=}e_1, \ldots, x_n{=}e_n\ \mathbf{in}\ e_{n+1} \Rightarrow e_i}$$

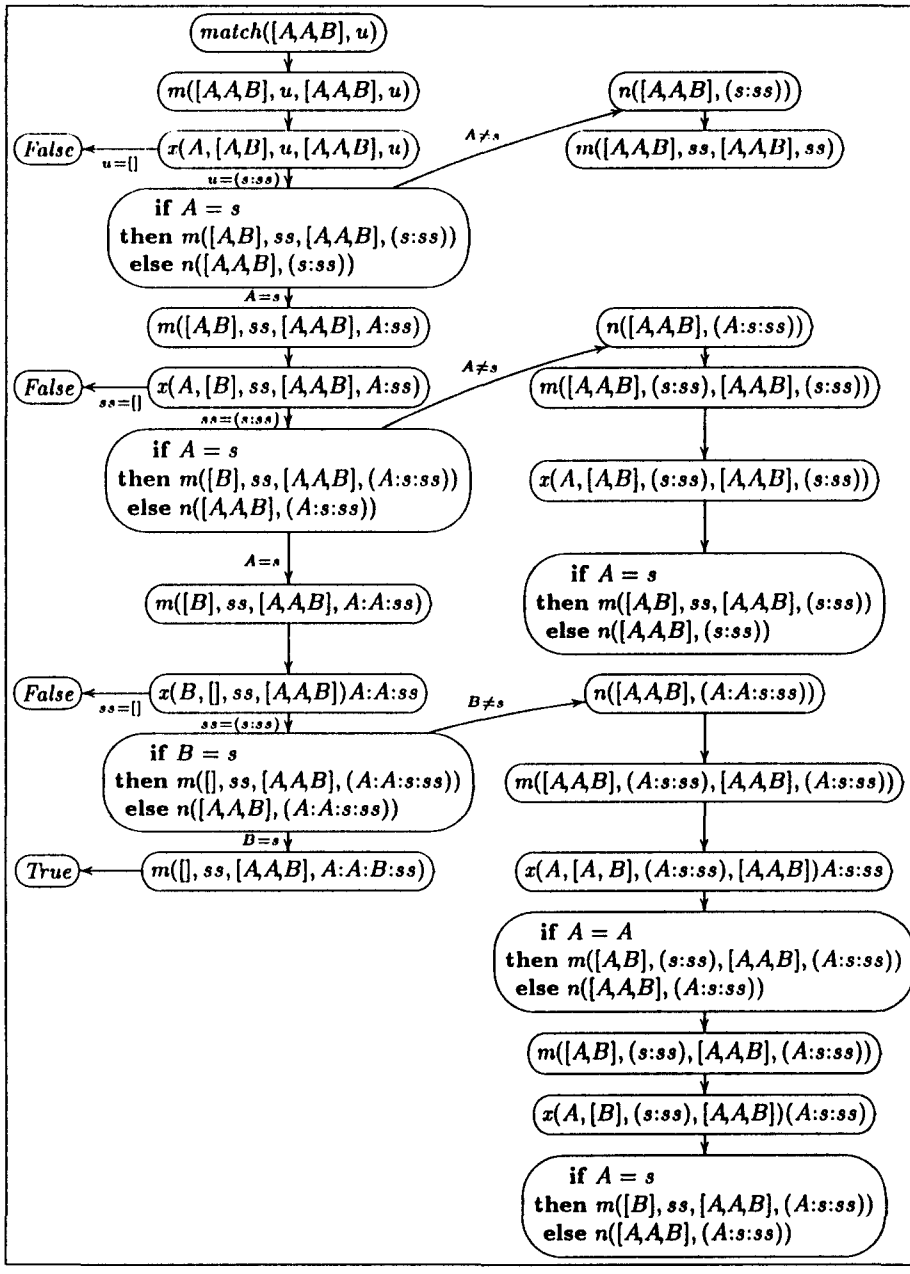


Fig. 2. Driving the naively specialized matcher

2. Let $t \in T(\mathcal{L})$ and $\beta \in \text{leaf}(t)$. Then $\text{drive}(t, \beta)$ is defined as in Definition 12.

Remark 6. The reason that only the relation $\Rightarrow$ is affected, not $\rightarrow_\theta$ and $\twoheadrightarrow_\theta$, is that let-expressions are allowed at the top-level of an expression only.

The new rule for $\Rightarrow$ expresses the semantics of generalizations: that we are trying to keep subexpressions apart.

Example 10. $\textbf{let } zs{=}xs \textbf{ in } a(a(xs, ys), zs) \Rightarrow a(a(xs, ys), zs)$.

Example 11. The driving steps in Example 2 are driving steps in the above sense.

6.2 Generalization Step

There are two types of generalization step in positive supercompilation: *abstract* and *split*. Both types of step are invoked when the expression of a leaf node is similar, in a certain sense, to an ancestor's expression.

The generalization step in Example 2 is an example of an abstract step. In this type of step 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. More precisely, this kind of step is an *upwards abstract* step.

In case the expression is an instance of the ancestor expression, we can replace the *leaf* node by a new node with an expression capturing the common structure. This type of step is called a *downwards abstract* step. For instance, if the leaf expression is $f([])$ and the ancestor expression is $f(xs)$, we can replace the leaf node by a node with expression $\textbf{let } xs = [] \textbf{ in } f(xs)$. By driving, this node will receive two children labeled $[]$ 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 be similar to an ancestor's expression, 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 would not make any progress towards termination of the supercompilation process. For instance, we might have a leaf with expression $f(g(x))$ and an ancestor with expression $g(x)$. We shall see later that these expressions will be counted as similar, but their msg is a variable. Therefore, applying an abstract step (upwards or downwards) would replace a node labeled e with a new node labeled $\textbf{let } z{=}e \textbf{ in } z$ which, by driving, would receive a child labeled e. Thus, no progress has been made. Instead, a split step is therefore performed. The idea behind a split step is that if the ancestor expression is similar to the leaf expression, then there is a subterm of the leaf expression which has non-trivial structure in common with the ancestor. Hence, the split step digs out this structure.

The following, then, are the generalization steps used in positive supercompilation; they are inspired by similar operations in [16]. The driving and generalization steps are illustrated in Figure 3.

Definition 17. Let $t \in T(\mathcal{L})$.

1. For $\beta \in \text{leaf}(t)$ with $t(\beta) = h(e_1, \ldots, e_n)$, $h \in C \cup F \cup G \cup \{\textbf{if}\}$, and $e_i \notin X$ for some $i \in \{1, \ldots, n\}$, define

$$\text{split}(t, \beta) = t\{\beta{:=}\textbf{let } x_1{=}e_1, \ldots, x_n{=}e_n \textbf{ in } h(x_1, \ldots, x_n) \rightarrow\}$$

2. For $\alpha, \beta \in \text{dom}(t)$, $t(\alpha), t(\beta) \in \mathcal{E}$ with $e_g = t(\alpha) \sqcap t(\beta)$ where $e_g \notin X$, $x_1, \ldots, x_n \in \text{vars}(e_g)$, $t(\alpha) = e_g\{x_1 := e_1, \ldots, x_n := e_n\}$, and $t(\alpha)$ not a renaming of e_g, define

$$\text{abstract}(t, \alpha, \beta) = t\{\alpha{:=}\textbf{let } x_1{=}e_1, \ldots, x_n{=}e_n \textbf{ in } e_g \rightarrow\}$$

Remark 7. Note that the above operations are allowed only under circumstances that guarantee that the constructed let-expression is well-formed according to the conditions of Definition 15.

Example 12. The generalization step in Example 2 is an abstract step.

6.3 When to Stop?

Above we described *how* to generalize in positive supercompilation. It remains to decide *when* to generalize, i.e., to decide when expressions are similar. The following relation $\trianglelefteq$ is used for that end.

Definition 18. The *homeomorphic embedding* $\trianglelefteq$ is the smallest relation on $\mathcal{E}_H(V)$ such that, for all $h \in H$, $x, y \in V$, and $e_i, e_i' \in \mathcal{E}_H(V)$:

$$x \trianglelefteq y \qquad \frac{\exists i \in \{1, \ldots, n\} : e \trianglelefteq e_i'}{e \trianglelefteq h(e_1', \ldots, e_n')} \qquad \frac{\forall i \in \{1, \ldots, n\} : e_i \trianglelefteq e_i'}{h(e_1, \ldots, e_n) \trianglelefteq h(e_1', \ldots, e_n')}$$

Example 13. The following expressions from $\mathcal{E}_H(V)$ give examples and non-examples of embedding, where $x, y \in V$, and $b, c, d, f \in H$.

$$\begin{array}{ll} b \trianglelefteq f(b) & f(c(b)) \not\trianglelefteq c(b) \\ c(b) \trianglelefteq c(f(b)) & f(c(b)) \not\trianglelefteq c(f(b)) \\ d(b, b) \trianglelefteq d(f(b), f(b)) & f(c(b)) \not\trianglelefteq f(f(f(b))) \end{array}$$

The rationale behind using the homeomorphic embedding relation in program transformers is that in any infinite sequence $e_0, e_1, \ldots$ of expressions, there *definitely* are $i < j$ with $e_i \trianglelefteq e_j$ (this property holds regardless of how the sequence $e_0, e_1, \ldots$ was produced and is known as *Kruskal's Theorem*—see e.g. [3]). 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.

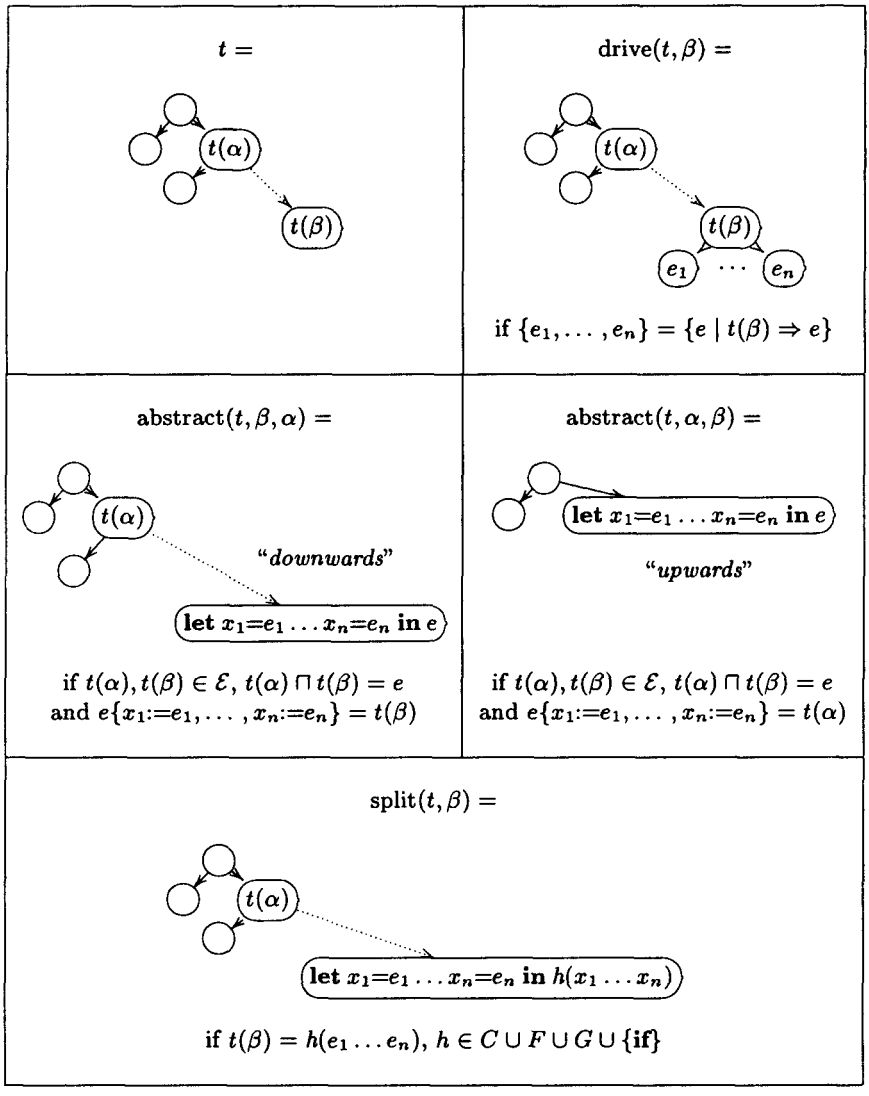


Fig. 3. Steps used in Positive Supercompilation

The homeomorphic embedding relation and the most specific generalization operation are defined on elements of $\mathcal{E}$, not on elements $\mathcal{L}$. Therefore, in order to compare nodes and compute new nodes in trees over $\mathcal{L}$, we have to either extend these operations to $\mathcal{L}$ or make sure that they are not applied to elements of $\mathcal{L}$. We choose the latter by always driving a node with a proper let-expression without comparing to ancestors. Also, when a node is compared to ancestors we do not compare it to those with proper let-expressions.

This idea—and related ideas discussed below—will be accommodated by comparing a leaf expression to the expressions of a certain *subset* of the ancestors which depends on the leaf node. This subset will be called the *relevant* ancestors. Thus, the relevant ancestors of a leaf with a proper let-expression is the empty set.

6.4 Transient Reductions

A *transient reduction* is a driving step applied to a node that will receive exactly one new child (the expression in the former node might aptly be called *deterministic.*) In some variants of supercompilation, transient reductions are performed without testing for similarity of ancestors. Therefore, transient reductions can go on indefinitely in certain cases.

Example 14. In Figure 2 we might perform the driving step at node

$$\boxed{m([A,A,B],(A{:}s{:}ss),[A,A,B],(A{:}s{:}ss))}$$

adding child

$$\boxed{\begin{array}{l}\textbf{if } A = A\\ \textbf{then } m([A,B],(s{:}ss),[A,A,B],(A{:}s{:}ss))\\ \textbf{else } n([A,A,B],(A{:}s{:}ss))\end{array}}$$

without testing whether any ancestors are embedded in the expression in the former node. In fact, there is an ancestor whose expression is embedded in the former node's expression. Thus, if the supercompilation algorithm with generalization does not incorporate some form of transient reduction, it will stop the development of the node and generalize prematurely. Hence, it will fail to handle the example satisfactorily.

The same holds for Example 1. Here the expression in node

$$\boxed{a(u{:}a(us,ys),zs)}$$

has an ancestor's expression embedded implying premature generalization unless some form of transient reductions is adopted.

We shall adopt a form of transient reductions that does not endanger termination of positive supercompilation and which is similar to *local unfolding* as adopted in partial deduction—see e.g. [15]. We will divide all nodes with non-proper let-expressions into two categories: global ones and local ones. The *global* nodes are those that give rise to instantiation of variables in driving steps. In other words, their reduction is non-transient. For instance, in Example 1, the nodes labeled $a(a(xs,ys),zs)$ and $a(ys,zs)$ are global. Conversely, the *local* nodes are the non-global ones. Local nodes give rise to transient reductions.

When considering a global leaf node we will compare it only to its global ancestors. When considering a local node, we will compare it only to its immediate local ancestors up to (but *not* including) the nearest global ancestor.

Definition 19. Let $t \in T(\mathcal{L})$ and $\beta \in \mathrm{dom}(t)$.

1. β is *proper* if $t(\beta)$ is a proper let-expression.
2. β is *global* if β is non-proper and $t(\beta) \rightarrow_\theta e$ for some $\theta \neq \{\}$. Also β is *local* if β is non-proper and not global.
3. The set of *immediate local ancestors* of β in t, $\mathrm{locanc}(t, \beta)$, is the set of local nodes in the longest branch $\alpha_1, \ldots, \alpha_n, \beta$ in t $(n \geq 0)$ such that $t(\alpha_1), \ldots, t(\alpha_n)$ are all local or proper.
4. The set of *global ancestors* of β in t, $\mathrm{globanc}(t, \beta)$, is the set of global nodes in $\mathrm{anc}(t, \beta)$.

In conclusion, a leaf node is driven when no relevant ancestor's expression is embedded in the leaf's expression. It remains to define the set of relevant ancestors.

Definition 20. Let $t \in T(\mathcal{L})$ and $\beta \in \mathrm{dom}(t)$. The set of *relevant ancestors* of β in t, $\mathrm{relanc}(t, \beta)$, is defined by:

$$\mathrm{relanc}(t, \beta) = \begin{cases} \{\} & \text{if } t(\beta) \text{ is proper} \\ \mathrm{locanc}(t, \beta) & \text{if } t(\beta) \text{ is local} \\ \mathrm{globanc}(t, \beta) & \text{if } t(\beta) \text{ is global} \end{cases}$$

6.5 Positive Supercompilation

First we adapt the notions of *processed node* and *closed tree* to trees over $\mathcal{L}$, then we define positive supercompilation $M_{ps} : \mathcal{E} \rightarrow T(\mathcal{L})$.

Definition 21. Let $t \in T(\mathcal{L})$. A $\beta \in \mathrm{leaf}(t)$ is *processed* if $t(\beta)$ is non-proper and one of the conditions in Definition 13 hold. Also, t is *closed* if all leaves in t are processed.

Definition 22. Let q be a program, and define $M_{ps} : \mathcal{E} \rightarrow T(\mathcal{L})$ by:

input $e \in \mathcal{E}$;
let $t = e\!\rightarrow$;
while t is not closed
  let $\beta \in \mathrm{leaf}(t)$ be an unprocessed node;
  if $\forall \alpha \in \mathrm{relanc}(t, \beta) \backslash \{\beta\} : t(\alpha) \not\trianglelefteq t(\beta)$ **then** $t = \mathrm{drive}(t, \beta)$
  else begin
    let $\alpha \in \mathrm{relanc}(t, \beta)$ and $t(\alpha) \trianglelefteq t(\beta)$.
    if $t(\alpha) \leq t(\beta)$ **then** $t = \mathrm{abstract}(t, \beta, \alpha)$
    else if $t(\alpha) \leftrightarrow t(\beta)$ **then** $t = \mathrm{split}(t, \beta)$
    else $t = \mathrm{abstract}(t, \alpha, \beta)$.
  end
return t;

Example 15. The algorithm computes exactly the trees in Examples 1 and 2, and the tree in Figure 2.

Remark 8. The algorithm calls abstract and split only in cases where these operations are well-defined.

As for termination, it is proven [24] that (a variant of) this positive supercompiler always terminates, i.e., the algorithm actually defines a total map.

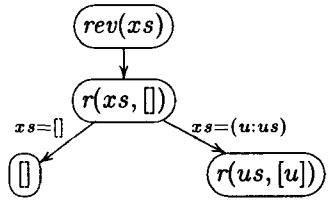
7 More Examples

This section illustrates how positive supercompilation works on a number of examples. Positive supercompilation makes no improvement on the programs; the interesting point is how termination is ensured. The first example covers accumulating parameters, the second example obstructing function calls.

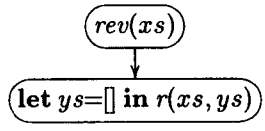
Example 16 (The accumulating parameter). Consider the following functional program reversing a list by means of an accumulating parameter:

$$\begin{array}{ll} rev(xs) & = r(xs, []) \\ r([], vs) & = vs \\ r(u{:}us, vs) & = r(us, u{:}vs) \end{array}$$

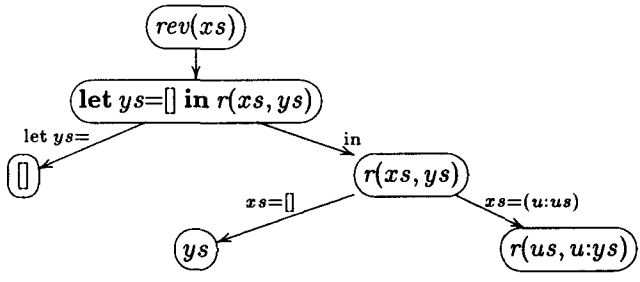
What happens if we apply M_{ps} to $rev(xs)$? After two driving steps we have:



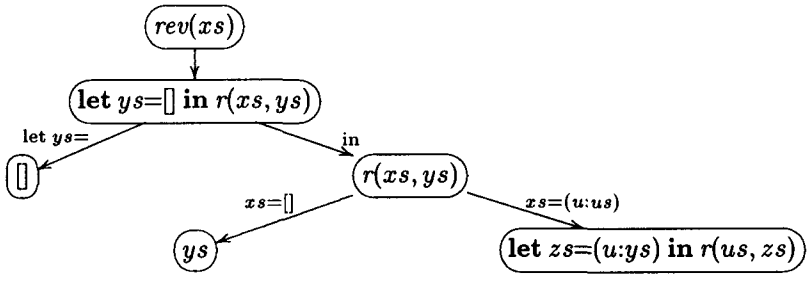
Then a generalization step yields:



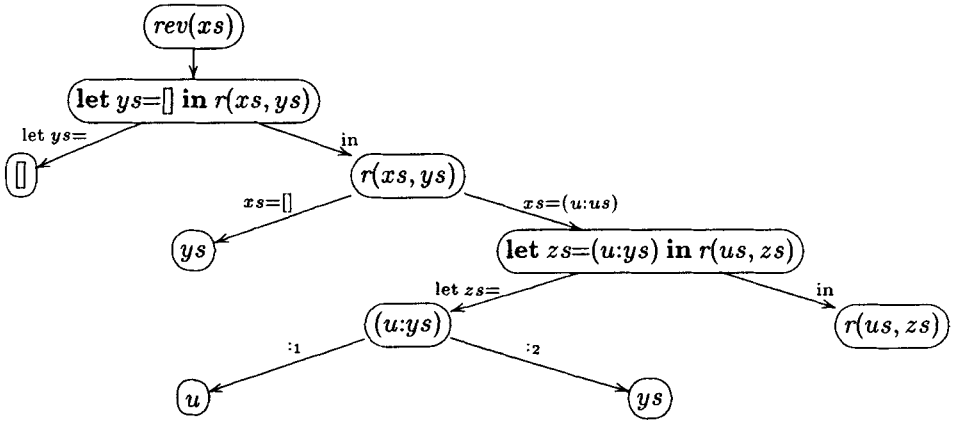
A few more driving steps yield:



Another generalization step yields:



Driving finally leads to the following closed tree:



From this tree one can construct a new term and program which turn out to be identical to the original term and program.

Example 17 (The obstructing function call). Consider another functional program reversing a list:

$$\begin{aligned} rev([]) &= [] \\ rev(u{:}us) &= a'(rev(us), u) \\ a'([], v) &= [v] \\ a'(u{:}us, v) &= u{:}a'(us, v) \end{aligned}$$

What happens if we apply M_{ps} to $rev(xs)$?

Example 18 (The accumulating side effect). Finally, consider the following functional program.

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

Here f is not intended to be an interesting function, merely to provide a simple illustration of a problem that occurs in more complicated contexts.

What happens if we apply M_{ps} to $f(v, v)$?

8 Discussion

We end the paper by briefly discussing correctness of the positive supercompiler and comparing it to other program transformers.

First of all, the driving algorithm in Section 4 is very similar to Wadler's *deforestation* [32]. This is perhaps not so surprising since we saw in Example 1 that positive supercompilation can eliminate intermediate data structures, which is the purpose of deforestation. The common features of the two transformers that facilitate elimination of intermediate data structures are that nested calls are kept together and that these are processed in normal order (outside-in).

In fact, the only essential difference between deforestation and the driving algorithm is in rule (11) in Figure 1: whereas the conclusion in the present rule is $e \Rightarrow e'\theta$, the corresponding rule in deforestation would be $e \Rightarrow e'$ (the substitution θ can then be omitted from the relations $\rightarrow_\theta$ and $\twoheadrightarrow_\theta$).

However, this difference has important consequences. For instance, the example transformation in Section 5 cannot be achieved by deforestation. Conversely, an important aspect of deforestation is that it terminates on all so-called *treeless* programs, and this does not hold for the driving algorithm. In other words, the increased power of driving comes at the expense of rarer termination, and this in turn is responsible for the fact that the problem of ensuring termination of positive supercompilation is quite difficult. This is elaborated at length in [23].

Positive supercompilation can also perform specialization like *partial evaluation.* However, most partial evaluators differ from positive supercompilation in their handling of nested calls which are processed in applicative order (inside-out) or not kept together. For this reason most partial evaluators are not able to eliminate intermediate data structures like lists—as is explained in [18].

Also, most partial evaluators do not perform *unification-based information propagation* [6, 9]; that is, there is the same difference between partial evaluators and positive supercompilation as there is between deforestation and positive supercompilation—the difference in rule (11) of driving. For this reason, partial evaluators typically cannot achieve the transformation in Section 5 without changes in the original matcher. On the other hand, the implementation aspects of partial evaluators are far more well-developed than those for supercompilers. An abstract framework for describing constant-based partial evaluation and driving is provided in [10].

In positive supercompilation, negative information (restrictions) are ignored when entering an else-branch of a conditional. *Perfect driving* [6] is a variant of supercompilation that propagates positive and negative information. Hence, the loss of information in supercompilation can be restricted to one phase in the transformer, namely generalization. However, generalization is not considered in [6].

Finally, positive supercompilation is related to *partial deduction*, e.g., [15, 16] and, more closely, to *conjunctive partial deduction* [5], as is developed at length in [8]. *Generalized partial computation* [4] has a similar effect and power as supercompilation, but requires the use of a theorem prover. A taxonomy of related transformers can be found in [9].

What does it mean that positive supercompilation is correct? There are three issues: preservation of semantics, non-degradation of efficiency, and termination.

As for *preservation of semantics,* the new program recovered from the tree produced by M_{ps} (if any) should be semantically equivalent to the original program. The main point is that the new program terminates neither more nor less than the original one. A general technique due to Sands [20] can be used to prove this for positive supercompilation [19].

As for *non-degradation in efficiency,* the output of M_{ps} should be at least as efficient as the input. There are several aspects of this problem.

First, there is the problem of avoiding *duplication of computation.* Since driving can cause function call duplication, a polynomial time program can be changed into an exponential time program. In deforestation this is avoided by considering only *linear* terms. This issue is beyond the scope of this paper.

Second, there is the problem of *code duplication.* Unrestrained unfolding may increase the size of a program dramatically. In principle, the size of a program does not degrade its efficiency. Again, this issue is beyond the scope of this paper.

As for *termination,* it is proven [24] that (a variant of) M_{ps} always terminates.

References

1. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
2. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
3. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3, 1987.
4. Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
5. R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling conjunctive partial deduction. In H. Kuchen and D.S. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1996.
6. R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
7. R. Glück and A.V. Klimov. A regeneration scheme for generating extensions. *Information Processing Letters*, 62(3):127–134, 1997.
8. R. Glück and M.H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Languages: Implementations, Logics and Programs*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, 1994.
9. R. Glück and M.H. Sørensen. A roadmap to metacomputation by supercompilation. In Danvy et al. [2], pages 137–160.
10. N.D. Jones. The essence of program transformation by partial evaluation and driving. In N.D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language, and Computation*, volume 792 of *Lecture Notes in Computer Science*, pages 206–224. Springer-Verlag, 1994. Festschrift in honor of S.Takasu.
11. A.V. Klimov and S.A. Romanenko. Metavychislitel' dlja jazyka Refal. Osnovnye ponjatija i primery. (A metaevaluator for the language Refal. Basic concepts and examples). Preprint 71, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987. (in Russian).
12. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
13. M. Krog and P. Rasmussen. Positive supercompilation. Manuscript, 1998.
14. J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

15. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Danvy et al. [2], pages 263–283.
16. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *International Conference on Logic Programming*, pages 597–613. MIT Press, 1995.
17. A.P. Nemytykh, V.A. Pinchuk, and V.F. Turchin. A self-applicable supercompiler. In Danvy et al. [2], pages 322–337.
18. K. Nielsen and M.H. Sørensen. Call-by-name CPS-translation as a binding-time improvement. In A. Mycroft, editor, *Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 296–313. Springer-Verlag, 1995.
19. D. Sands. Proving the correctness of recursion-based automatic program transformation. In P. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 681–695. Springer-Verlag, 1995.
20. D. Sands. Total correctness by local improvement in program transformation. In *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232. ACM Press, 1995.
21. M.H. Sørensen. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.
22. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
23. M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
24. M.H.B. Sørensen. Convergence of program transformers in the metric space of trees. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 315–337. Springer-Verlag, 1998.
25. V.F. Turchin. *The Phenomenon of Science.* Columbia University Press, New York, 1977.
26. V.F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, 1979.
27. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
28. V.F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
29. V.F. Turchin. Metacomputation: Metasystem transition plus Supercompilation. In Danvy et al. [2], pages 481–510.
30. V.F. Turchin, R. Nirenberg, and D. Turchin. Experiments with a supercompiler. In *ACM Conference on Lisp and Functional Programming*, pages 47–55. ACM Press, 1982.
31. W. Vanhoof. Implementatie van een supercompilator voor een functionele taal (in dutch). Master's thesis, University of Leuven, 1996.
32. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.