

# Convergence of Program Transformers in the Metric Space of Trees

Morten Heine B. Sørensen

Department of Computer Science, University of Copenhagen (DIKU)  
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark  
E-mail: rambo@diu.dk

**Abstract.** In recent years increasing consensus has emerged that program transformers, e.g., partial evaluation and unfold/fold transformations, should terminate; a compiler should stop even if it performs fancy optimizations! A number of techniques to ensure termination of program transformers have been invented, but their correctness proofs are sometimes long and involved.

We present a framework for proving termination of program transformers, cast in the *metric space of trees*. We first introduce the notion of an *abstract program transformer*; a number of well-known program transformers can be viewed as instances of this notion. We then formalize what it means that an abstract program transformer *terminates* and give a general *sufficient condition* for an abstract program transformer to terminate. We also consider some *specific techniques* for satisfying the condition. As *applications* we show that termination of some well-known program transformers either follows directly from the specific techniques or is easy to establish using the general condition.

Our framework facilitates simple termination proofs for program transformers. Also, since our framework is independent of the language being transformed, a single correctness proof can be given in our framework for program transformers using essentially the same technique in the context of different languages. Moreover, it is easy to extend termination proofs for program transformers to accommodate changes to these transformers. Finally, the framework may prove useful for designing new termination techniques for program transformers.

## 1 Introduction

Numerous program transformation techniques have been studied in the areas of functional and logic languages, e.g., partial evaluation and unfold/fold transformations. Pettorossi and Proietti [30] show that many of these techniques can be viewed as consisting of three conceptual phases which may be interleaved: *symbolic computation*, *search for regularities*, and *program extraction*.

Given a program, the first phase constructs a possibly infinite tree in which each node is labeled with an expression; children are added to the tree by *unfolding* steps. The second phase employs *generalization* steps to ensure that one constructs a finite tree. The third phase constructs from this finite tree a new program.

The most difficult problem for most program transformers is to formulate the second phase in such a way that the transformer both performs interesting optimizations and always terminates. Solutions to this problem now exist for most transformers.

The proofs that these transformers indeed terminate—including some proofs by the author—are sometimes long, involved, and read by very few people. One reason for this

is that such a proof needs to formalize what it means that the transformer terminates, and significant parts of the proof involve abstract properties about the formalization.

In this paper we present a framework for proving termination of program transformers. We first introduce the notion of an *abstract program transformer*, which is a map from trees to trees expressing one step of transformation. A number of well-known program transformers can be viewed as instances of this notion. Indeed, using the notion of an abstract program transformer and associated general operations on trees, it is easy to specify and compare various transformers, as we shall see.

We then formalize what it means that an abstract program transformer *terminates* and give a *sufficient condition* for an abstract program transformer to terminate. A number of well-known transformers satisfy the condition. In fact, termination proofs for some of these transformers implicitly contain the correctness proof of the condition. Developing the condition once and for all factors out this common part; a termination proof within our framework for a program transformer only needs to prove properties that are specific to the transformer. This yields shorter, less error-prone, and more transparent proofs, and means that proofs can easily be extended to accommodate changes in the transformer. Also, our framework isolates exactly those parts of a program transformer relevant for ensuring termination, and this makes our framework useful for designing new termination techniques for existing program transformers.

The insight that various transformers are very similar has led to the exchange of many ideas between researchers working on different transformers, especially techniques to ensure termination. Variations of one technique, used to ensure termination of positive supercompilation [35], have been adopted in partial deduction [23], conjunctive partial deduction [16], Turchin’s supercompiler [41], and partial evaluation of functional-logic programs [1]. While the technique is fairly easily transported between different settings, a separate correctness proof has been given in each setting.

It would be better if one could give a single proof of correctness for this technique in a setting which abstracts away irrelevant details of the transformers. Therefore, we consider *specific techniques*, based on well-known transformers, for satisfying the condition in our framework. The description of these techniques is *specific* enough to imply termination of well-known transformers, and *general* enough to establish termination of different program transformers using essentially the same technique in the context of different languages. As *applications* we demonstrate that this is true for positive supercompilation and partial deduction (in the latter case by a brief sketch).

The set of trees forms a metric space, and our framework can be elegantly presented using such notions as convergence and continuity in this metric space. We also use a few well-known results about the metric space of trees, e.g., completeness. However, we do not mean to suggest that the merits of our approach stem from the supposed depth of any of these results; rather, the metric space of trees offers concepts and terminology useful for analyzing termination of abstract program transformers.

Section 2 introduces program transformers as maps from trees to trees. This is then formalized in the notion of an abstract program transformer in Section 3. Section 4 presents positive supercompilation as an abstract program transformer. Section 5 presents the metric space of trees, and Section 6 uses this to present our sufficient condition for termination, as well as the specific techniques to satisfy the condition. Section 7 shows that positive supercompilation terminates. It also sketches Martens and Gallagher’s [26] generic algorithm for partial deduction as an abstract program

transformer and sketches a proofs that it terminates.

We stress that it is not the intention of this paper to advocate any particular technique that ensures termination of program transformers; rather, we are concerned with a general method to prove that such techniques are correct.

This work is part of a larger effort to understand the relation between deforestation, supercompilation, partial deduction, and other program transformers better [17, 18, 20, 36, 37] and to develop a unifying theory for such transformers.

## 2 Trees in Transformation

We now proceed to show how program transformers may be viewed as maps that manipulate certain trees, following Pettorossi and Proietti [30].

*Example 1.* Consider a functional program appending two lists.

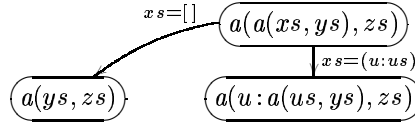
$$\begin{aligned} a([], vs) &= vs \\ a(u:us, vs) &= u : a(us, vs) \end{aligned}$$

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.

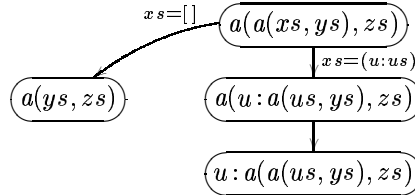
We begin with a tree whose single node is labeled with  $a(a(xs, ys), zs)$ :

$$\boxed{a(a(xs, ys), zs)}$$

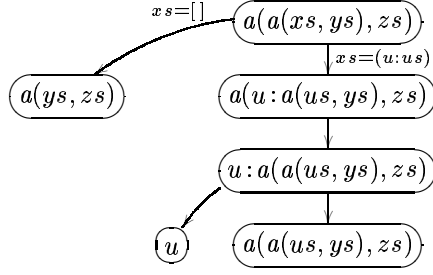
By an *unfolding* step which replaces the inner call to append according to the different patterns in the definition of  $a$ , two new expressions are added as labels on children:



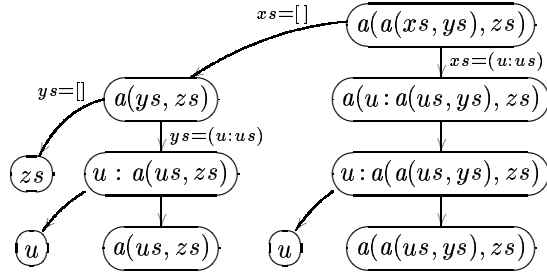
In the rightmost child we can perform an unfolding step which replaces the outer call to append:



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. Unfolding 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 free variables of  $e$ .

To construct a new program from a closed tree, we introduce, roughly, for each node  $\alpha$  with child  $\beta$  a definition where the left and right hand side of the definition are derived from  $\alpha$  and  $\beta$ , respectively. More specifically, in the above example we rename expressions of form  $a(a(xs, ys), zs)$  as  $aa(xs, ys, zs)$ , 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  $a(a(xs, ys), zs)$  in the original program, since the new expression traverses  $xs$  only once.

The transformation in Example 1 proceeded in three phases—symbolic computation, search for regularities, and program extraction—the first two of which were

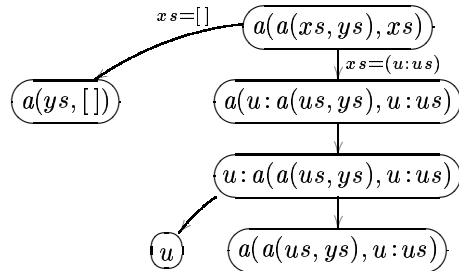
interleaved. In the first phase we performed unfolding steps that added children to the tree. In the second phase we made sure that no node with an expression which was a renaming of ancestor's expression was unfolded, and we continued the overall process until the tree was closed. In the third phase 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.

*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:

$$a(a(xs, ys), xs)$$

After the first few steps we have:

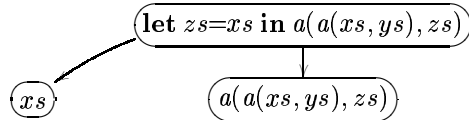


Unlike the situation in Example 1, the label of the rightmost node is not a renaming of the expression at the root. In fact, repeated unfolding will *never* lead to that situation; special measures must be taken.

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:

$$\text{let } zs=xs \text{ in } a(a(xs, ys), zs)$$

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



Unfolding of the node labeled  $a(a(xs, ys), zs)$  leads to the same tree as 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.<sup>1</sup>

<sup>1</sup> 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.

Numerous program transformers can be cast more or less accurately in the three above mentioned phases, e.g., partial deduction [23, 26], conjunctive partial deduction [16], compiling control [10], loop absorption [31], partial evaluation of functional-logic languages [1], unfold/fold transformation of functional programs [11], unfold/fold transformation of logic programs [38], tupling [4, 29], supercompilation [39, 40], positive supercompilation [18, 35], generalized partial computation [15], deforestation [42], and online partial evaluation of functional programs [43, 33, 21].

Although *offline* transformers (i.e., transformers making use of analyses prior to the transformation to make changes in the program ensuring termination) may fit into the description with the three phases, the second phase is rather trivial, amounting to the situation in Example 1.

### 3 Abstract Program Transformers

We now formalize the idea that a program transformer is a map from trees to trees, expressing one step of transformation. We first introduce trees in a rigorous manner, following Courcelle [12].

**Definition 1.** A *tree* over a set  $E$  is a partial map<sup>2</sup>  $t : \mathbb{N}_1^* \rightarrow E$  such that

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

Let  $t$  be a tree over  $E$ . The elements of  $\text{dom}(t)$  are called *nodes* of  $t$ ; the empty string  $\epsilon$  is the *root*, and for any node  $\alpha$  in  $t$ , the nodes  $\alpha i$  of  $t$  (if any) are the *children* of  $\alpha$ , and we also say that  $\alpha$  is the *parent* of these nodes. A *branch* in  $t$  is a finite or infinite sequence  $\alpha_0, \alpha_1, \dots \in \text{dom}(t)$  where  $\alpha_0 = \epsilon$  and, for all  $i$ ,  $\alpha_{i+1}$  is a child of  $\alpha_i$ . A node with no children is a *leaf*. We denote by  $\text{leaf}(t)$  the set of all leaves in  $t$ . For any node  $\alpha$  of  $t$ ,  $t(\alpha) \in E$  is the *label* of  $\alpha$ . Also,  $t$  is *finite*, if  $\text{dom}(t)$  is finite. Finally,  $t$  is *singleton* if  $\text{dom}(t) = \{\epsilon\}$ , i.e., if  $\text{dom}(t)$  is singleton.

$T_\infty(E)$  is the set of all trees over  $E$ , and  $T(E)$  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  $x, xs, \dots \in V$  and  $a, \text{cons}, \text{nil} \in H$ , denoting  $(x : xs)$  by  $\text{cons}(x, xs)$  and  $[]$  by  $\text{nil}$ . Then let  $\mathcal{L}_H(V)$  be the smallest set such that  $e_1, \dots, e_n, e \in \mathcal{E}_H(V)$  implies that **let**  $x_1=e_1, \dots, x_n=e_n$  **in**  $e \in \mathcal{L}_H(V)$ . The trees in Example 1 and 2 (ignoring labels on edges) are a diagrammatical presentation of trees over  $\mathcal{E}_H(V)$  and  $\mathcal{L}_H(V)$ , respectively.

**Definition 2.** An *abstract program transformer* (for brevity also called an *apt*) on  $E$  is a map  $M : T(E) \rightarrow T(E)$ .

<sup>2</sup> We let  $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$ .  $S^*$  is the set of finite strings over  $S$ , and  $\text{dom}(f)$  is the domain of a partial function  $f$ .

For instance, the sequences of trees in Example 1 and 2 could be computed by iterated application of some apt. How do we formally express that no more transformation steps will happen, i.e., that the apt has produced its final result? In this case,  $M$  returns its argument tree unchanged, i.e.,  $M(t) = t$ .

**Definition 3.**

1. An apt  $M$  on  $E$  *terminates on*  $t \in T(E)$  if  $M^i(t) = M^{i+1}(t)$  for some  $i \in \mathbb{N}$ .<sup>3</sup>
2. An apt  $M$  on  $E$  *terminates* if  $M$  terminates on all singletons  $t \in T(E)$ .

Although apt's are defined on the set  $T(E)$  of finite trees, it turns out to be convenient to consider the general set  $T_\infty(E)$  of finite as well as infinite trees.

The rest of this section introduces some definitions pertaining to trees that will be used in the remainder.

**Definition 4.** Let  $E$  be a set, and  $t, t' \in T_\infty(E)$ .

1. The *depth*  $|\alpha|$  of a node  $\alpha$  in  $t$  is:

$$\begin{aligned} |\epsilon| &= 0 \\ |\alpha i| &= |\alpha| + 1 \end{aligned}$$

2. The *depth*  $|t|$  of  $t$  is defined by:

$$|t| = \begin{cases} \max\{|\alpha| \mid \alpha \in \text{dom}(t)\} & \text{if } t \text{ is finite} \\ \infty & \text{otherwise} \end{cases}$$

3. The *initial subtree of depth  $\ell$  of  $t$* , written  $t[\ell]$ , is the tree  $t'$  with

$$\begin{aligned} \text{dom}(t') &= \{\alpha \in \text{dom}(t) \mid |\alpha| \leq \ell\} \\ t'(\alpha) &= t(\alpha) \quad \text{for all } \alpha \in \text{dom}(t') \end{aligned}$$

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

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

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

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

7. We denote by  $e \rightarrow e_1, \dots, e_n$  the tree  $t \in T_\infty(E)$  with

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

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

---

<sup>3</sup> For  $f : A \rightarrow A$ ,  $f^0(a) = a$ ,  $f^{i+1}(a) = f^i(f(a))$ .

In the diagrammatical notation of Section 2, the depth of a node is the number of edges on the path from the root to the node. The depth of a tree is the maximal depth of any node. The initial subtree of depth  $\ell$  is the tree obtained by deleting all nodes of depth greater than  $\ell$  and edges into such nodes. The tree  $t\{\alpha:=t'\}$  is the tree obtained by replacing the subtree with root  $\alpha$  in  $t$  by the tree  $t'$ . The ancestors of a node are the nodes on the path from the root to the node. Finally, the tree  $e \rightarrow e_1, \dots, e_n$  is the tree with root labeled  $e$  and  $n$  children labeled  $e_1, \dots, e_n$ , respectively.

## 4 Example: Positive Supercompilation

We present a variant of positive supercompilation [18, 35, 36, 37] as an abstract program transformer. We consider the following first-order functional language; the intended operational semantics is normal-order graph reduction to weak head normal form.

**Definition 5.** 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  $g \in G$ ; symbols all 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:

$$\begin{aligned}
\mathcal{Q} \ni q &::= d_1 \dots d_m \\
\mathcal{D} \ni d &::= f(x_1, \dots, x_n) \triangleq e && \text{(f-function)} \\
& \quad | \quad g(p_1, x_1, \dots, x_n) \triangleq e_1 && \text{(g-function)} \\
& \quad \quad \vdots \\
& \quad \quad g(p_m, x_1, \dots, x_n) \triangleq e_m \\
\mathcal{E} \ni e &::= x && \text{(variable)} \\
& \quad | \quad c(e_1, \dots, e_n) && \text{(constructor)} \\
& \quad | \quad f(e_1, \dots, e_n) && \text{(f-function call)} \\
& \quad | \quad g(e_0, e_1, \dots, e_n) && \text{(g-function call)} \\
\mathcal{P} \ni p &::= c(x_1, \dots, x_n)
\end{aligned}$$

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 on the right side of a definition be present in its left side. By  $\text{vars}(e)$  we denote the set of variables occurring in the expression  $e$ .

*Example 4.* The programs in Example 1–2 are programs in this language using the short notation  $[]$  and  $(x : xs)$  for the list constructors *nil* and *cons*( $x, xs$ ).

**Definition 6.** A *substitution* on  $\mathcal{E}_H(V)$  is a total map from  $V$  to  $\mathcal{E}_H(V)$ . We denote by  $\{x_1:=e_1, \dots, x_n:=e_n\}$  the substitution that maps  $x_i$  to  $e_i$  and all other variables to themselves. Substitutions are lifted to expressions as usual, and application of substitutions is written postfix.

For a substitution  $\theta$ ,  $\text{base}(\theta) = \{x \in X \mid x\theta \neq x\}$ . A substitution  $\theta$  is *free for* an  $e \in \mathcal{E}_H(V)$  if for all  $x \in \text{base}(\theta)$ :  $\text{vars}(x\theta) \cap \text{vars}(e) = \emptyset$ .

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



**Definition 7.** The set  $\mathcal{L}$  of let-expressions is defined as follows:

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

where  $n \geq 0$ . If  $n > 0$  then we require that  $x_1, \dots, x_n \in \text{vars}(e)$ , that  $e \notin X$ , and that  $e\{x_1 := e_1, \dots, x_n := e_n\}$  is not a renaming<sup>4</sup> of  $e$ . If  $n = 0$  then we identify the expression  $\mathbf{let} \ x_1=e_1, \dots, x_n=e_n \ \mathbf{in} \ e$  with  $e$ . Thus,  $\mathcal{E}$  is a subset of  $\mathcal{L}$ .

*Remark.* There is a close relationship between the set  $\mathcal{E}$  of expressions introduced above and the set  $\mathcal{E}_H(V)$  introduced in Example 3. In fact,  $\mathcal{E} = \mathcal{E}_{C \cup F \cup G}(X)$ . Therefore, in what follows we can make use of well-known facts about  $\mathcal{E}_H(V)$  in reasoning about  $\mathcal{E}$ . Also,  $\mathcal{L} = \mathcal{L}_{C \cup F \cup G}(X)$ .

We now set out to formulate the unfolding and generalization operations mentioned in Section 2, as used in positive supercompilation. We begin with unfolding.

The following relation  $\Rightarrow$  generalizes the small-step semantics for normal-order reduction to weak head normal form by propagating to the arguments of constructors and by working on expressions with variables; the latter is done by propagating unifications representing the assumed outcome of tests on constructors—notice the substitution  $\{y := p\}$  in the third rule. Also, the reduction for let-expressions expresses the semantics of generalizations: that we are trying to keep things apart.

**Definition 8.** For a program  $q$ , the relations  $e \rightarrow_\theta e'$  and  $\ell \Rightarrow e$  where  $e, e' \in \mathcal{E}$ ,  $\ell \in \mathcal{L}$ , and  $\theta$  is a substitution on  $\mathcal{E}$ , is defined by:

$$\begin{array}{c} \frac{f(x_1, \dots, x_n) \triangleq e \in q}{f(e_1, \dots, e_n) \rightarrow_{\{\}} e\{x_1 := e_1, \dots, x_n := e_n\}} \\[10pt] \frac{g(c(x_1, \dots, x_m), x_{m+1}, \dots, x_n) \triangleq e \in q}{g(c(e_1, \dots, e_m), e_{m+1}, \dots, e_n) \rightarrow_{\{\}} e\{x_1 := e_1, \dots, x_n := e_n\}} \\[10pt] \frac{g(p, x_1, \dots, x_n) \triangleq e \in q}{g(y, e_1, \dots, e_n) \rightarrow_{\{y:=p\}} e\{x_1 := e_1, \dots, x_n := e_n\}} \\[10pt] \frac{e \rightarrow_\theta e' \quad \& \quad \theta \text{ is free for } g(e, e_1, \dots, e_n)}{g(e, e_1, \dots, e_n) \rightarrow_\theta g(e', e_1, \dots, e_n)} \\[10pt] \frac{e \rightarrow_\theta e'}{e \Rightarrow e'\theta} \\[10pt] \frac{i \in \{1, \dots, n\}}{c(e_1, \dots, e_n) \Rightarrow e_i} \\[10pt] \frac{i \in \{1, \dots, n+1\}}{\mathbf{let} \ x_1=e_1, \dots, x_n=e_n \ \mathbf{in} \ e_{n+1} \Rightarrow e_i} \end{array}$$

The unfolding operation in positive supercompilation is called *driving*.

---

<sup>4</sup> The notion of a renaming is defined below.

**Definition 9.** Let  $t \in T(\mathcal{L})$  and  $\beta \in \text{leaf}(t)$ . Then

$$\text{drive}(t, \beta) = t\{\beta := t(\beta) \rightarrow e_1, \dots, e_n\}$$

where<sup>5</sup>  $\{e_1, \dots, e_n\} = \{e \mid t(\beta) \Rightarrow e\}$ .

*Example 5.* All the unfolding steps in Examples 1–2 are, in fact, driving steps.

Next we set out to formulate the generalization operations used in positive supercompilation. In generalization steps one often compares two expressions and extracts some common structure; the *most specific generalization*, defined next, extracts the most structure in a certain sense.

**Definition 10.** Let  $e_1, e_2 \in \mathcal{E}_H(V)$ , for some  $H, V$ .

1. The expression  $e_2$  is an *instance* of  $e_1$ ,  $e_1 \leq e_2$ , if  $e_1\theta = e_2$  for a substitution  $\theta$ .
2. The expression  $e_2$  is a *renaming* of  $e_1$  if  $e_1 \leq e_2$  and  $e_2 \leq e_1$ .
3. A *generalization* of  $e_1, e_2$  is an expression  $e_g$  such that  $e_g \leq e_1$  and  $e_g \leq e_2$ .
4. A *most specific generalization* (msg)  $e_1 \sqcap e_2$  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  $e'_g \leq e_g$ .
5. Two expressions  $e_1$  and  $e_2$  are *incommensurable*,  $e_1 \nleftrightarrow e_2$ , if  $e_1 \sqcap e_2$  is a variable.

*Example 6.* The following table gives example 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  $x, y \in V$ ,  $b, c, d, f \in H$ .

| $e_1$     | $e_2$           | $e_1 \sqcap e_2$ | $\theta_1$   | $\theta_2$      |
|-----------|-----------------|------------------|--------------|-----------------|
| $b$       | $f(b)$          | $x$              | $\{x := b\}$ | $\{x := f(b)\}$ |
| $c(b)$    | $c(f(b))$       | $c(x)$           | $\{x := b\}$ | $\{x := f(b)\}$ |
| $c(y)$    | $c(f(y))$       | $c(y)$           | $\{\}$       | $\{y := f(y)\}$ |
| $d(b, b)$ | $d(f(b), f(b))$ | $d(x, x)$        | $\{x := b\}$ | $\{x := f(b)\}$ |

**Proposition 11.** Let  $H, V$  be some sets. For all  $e_1, e_2 \in \mathcal{E}_H(V)$  there is an msg which is unique up to renaming.<sup>6</sup>

The following, then, are the generalization operations used in positive supercompilation; the operations are illustrated (together with driving) in Fig. 1.

**Definition 12.** Let  $t \in T(\mathcal{L})$ .

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

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

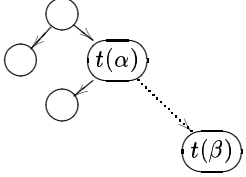
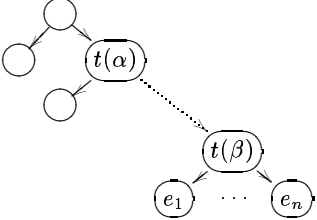
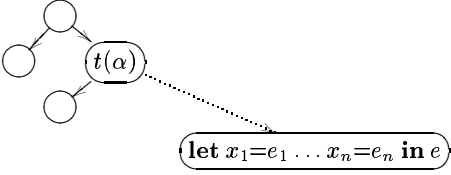
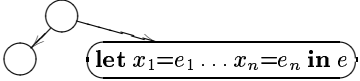
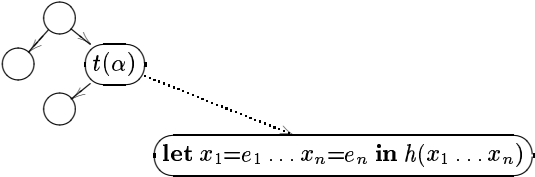
<sup>5</sup> We may have  $n \geq 2$  only when  $t(\beta)$  contains an outermost constructor of arity  $n$  with  $n \geq 2$ , and when  $t(\beta)$  contains a call to a g-function defined by  $n$  patterns with  $n \geq 2$ . For code generation purposes it is necessary in these cases to recall which constructor argument or which pattern each of the children  $e_1, \dots, e_n$  corresponds to, but this issue is ignored in the present paper.

<sup>6</sup> As a matter of technicality, we shall require that if  $e_1 \leq e_2$  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  $e_1$ .

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

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

*Remark.* 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 7.

|  |  |
|--|--|
| <p style="text-align: center;"><math>t =</math></p>   | <p style="text-align: center;"><math>\text{drive}(t, \beta) =</math></p>  <p style="text-align: center;">if <math>\{e_1, \dots, e_n\} = \{e \mid t(\beta) \Rightarrow e\}</math></p>   |
| <p style="text-align: center;"><math>\text{abstract}(t, \beta, \alpha) =</math></p>  <p style="text-align: center;">if <math>t(\alpha), t(\beta) \in \mathcal{E}</math>, <math>t(\alpha) \sqcap t(\beta) = e</math><br/>and <math>e\{x_1 := e_1, \dots, x_n := e_n\} = t(\beta)</math></p> | <p style="text-align: center;"><math>\text{abstract}(t, \alpha, \beta) =</math></p>  <p style="text-align: center;">if <math>t(\alpha), t(\beta) \in \mathcal{E}</math>, <math>t(\alpha) \sqcap t(\beta) = e</math><br/>and <math>e\{x_1 := e_1, \dots, x_n := e_n\} = t(\alpha)</math></p> |
| <p style="text-align: center;"><math>\text{split}(t, \beta) =</math></p>  <p style="text-align: center;">if <math>t(\beta) = h(e_1 \dots e_n)</math>, <math>h \in C \cup F \cup G</math></p>   |  |

**Fig. 1.** Operations used in Positive Supercompilation

*Example 7.* The generalization step in Example 2 is an abstract step.

This completes the description of the unfolding and generalization operations in positive supercompilation. It remains to decide *when* to generalize. The following relation  $\sqsubseteq$  is used for that end.

**Definition 13.** The *homeomorphic embedding*  $\sqsubseteq$  is the smallest relation on  $\mathcal{E}_H(V)$  such that, for all  $h \in H$ ,  $x, y \in V$ , and  $e_i, e'_i \in \mathcal{E}_H(V)$ :

$$x \sqsubseteq y \quad \frac{\exists i \in \{1, \dots, n\} : e \sqsubseteq e'_i}{e \sqsubseteq h(e'_1, \dots, e'_n)} \quad \frac{\forall i \in \{1, \dots, n\} : e_i \sqsubseteq e'_i}{h(e_1, \dots, e_n) \sqsubseteq h(e'_1, \dots, e'_n)}$$

*Example 8.* The following expressions from  $\mathcal{E}_H(V)$  give examples and non-examples of embedding, where  $x, y \in V$ , and  $b, c, d, f \in H$ .

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

The rationale behind using the homeomorphic embedding relation in program transformers is that in any infinite sequence  $e_0, e_1, \dots$  of expressions, there *definitely* are  $i < j$  with  $e_i \sqsubseteq e_j$ .<sup>7</sup> Thus, if unfolding is stopped at any node with an expression in which an ancestor's expression is embedded, unfolding cannot construct an infinite branch. Conversely, if  $e_i \sqsubseteq e_j$  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 unfolding is stopped for a good reason.

In some cases it is desirable to unfold a node even if its label expression has some ancestor's expression embedded. In the variant of positive supercompilation studied in this paper, this is done in two situations; the first is when the expression is *trivial*.

**Definition 14.** An element of  $\mathcal{L}$  is *trivial* if it has one of the following forms:

1. **let**  $x_1=e_1, \dots, x_m=e_m$  **in**  $e$  where  $m > 0$ ;
2.  $c(e_1, \dots, e_n)$ ;
3.  $x$ ;

New leaf expressions, resulting from driving a node with a trivial expression, are strictly smaller than the former expression, in a certain order.

*Remark.* All non-trivial elements of  $\mathcal{L}$  are, in fact, elements of  $\mathcal{E} \setminus X$ . Thus, the most specific generalization operation and the homeomorphic embedding relation which are defined on  $\mathcal{E}_H(V)$ —but not on  $\mathcal{L}_H(V)$ —apply to all non-trivial expressions. In particular,  $\text{abstract}(t, \alpha, \beta)$  and  $\text{split}(t, \beta)$  will be used only when  $t(\alpha)$  and  $t(\beta)$  are non-trivial.

The second situation in which we will unfold a node despite the fact that its expression has an ancestor's expression embedded, is when the first expression gave rise to several children corresponding to different patterns (as  $a(a(xs, ys), zs)$ ) whereas the

<sup>7</sup> This property holds regardless of how the sequence  $e_0, e_1, \dots$  was produced—see Theorem 41.

new expression does not give rise to several children according to different patterns (as in  $a(u : a(us, ys), zs)$ ). In such cases, new information is available in the new expression, and it is desirable that this be taken into account by an unfolding step.

This idea is formalized by the following map  $B$ , which gives a *very* simple version of the *characteristic trees*, studied by Leuschel and Martens [23] and others.

**Definition 15.** Define  $B : \mathcal{E} \rightarrow \mathbb{B}$  by

$$\begin{aligned} B(g(e_0, e_1, \dots, e_m)) &= B(e_0) \\ B(f(e_1, \dots, e_m)) &= 0 \\ B(c(e_1, \dots, e_m)) &= 0 \\ B(x) &= 1 \end{aligned}$$

We write  $e \trianglelefteq^* e'$  iff  $e \trianglelefteq e'$  and  $B(e) = B(e')$ .

This gives us enough terminology to explain when a leaf node should be driven: if its label is trivial, or if its label is non-trivial and no ancestor has a non-trivial label which is embedded with respect to  $\trianglelefteq^*$  in the leaf's label.

To formulate positive supercompilation we finally need to express when a node needs no further processing. The following will be used for that.

**Definition 16.** Let  $t \in T_\infty(\mathcal{L})$ . 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  $\alpha \in \text{anc}(t, \beta) \setminus \{\beta\}$  such that  $t(\alpha)$  is non-trivial and a renaming of  $t(\beta)$ .

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

Positive supercompilation  $M_{ps} : T(\mathcal{L}) \rightarrow T(\mathcal{L})$  can then be defined as follows.<sup>8</sup>

**Definition 17.** Given  $t \in T(\mathcal{L})$ , if  $t$  is closed  $M_{ps}(t) = t$ . Otherwise, let  $\beta \in \text{leaf}(t)$  be an unprocessed node and proceed as follows.

```

if  $t(\beta)$  is trivial,
or  $t(\beta)$  is non-trivial and  $\forall \alpha \in \text{anc}(t, \beta) \setminus \{\beta\} : t(\alpha) \text{ non-trivial} \Rightarrow t(\alpha) \not\trianglelefteq^* t(\beta)$ 
  then  $M_{ps}(t) = \text{drive}(t, \beta)$ 
else begin
  let  $\alpha \in \text{anc}(t, \beta)$ ,  $t(\alpha), t(\beta)$  be non-trivial, and  $t(\alpha) \trianglelefteq^* t(\beta)$ .
  if  $t(\alpha) \leq t(\beta)$ 
    then  $M_{ps}(t) = \text{abstract}(t, \beta, \alpha)$ 
  else if  $t(\alpha) \leftrightarrow t(\beta)$ 
    then  $M_{ps}(t) = \text{split}(t, \beta)$ 
  else  $M_{ps}(t) = \text{abstract}(t, \alpha, \beta)$ .
end

```

*Example 9.* The algorithm computes exactly the sequences of trees in Examples 1–2.

---

<sup>8</sup> 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.

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

The above algorithm is not the simplest conceivable version of positive supercompilation; indeed, from the point of view of termination it is somewhat involved. In Section 7 we prove that it terminates.

## 5 The Metric Space of Trees

As suggested by the examples in Section 2, termination of an online program transformer amounts to a certain form of convergence of sequences of trees. We now review some fundamental definitions and properties from the theory of metric spaces, which is a general framework for the study of convergence—see, e.g., [32]. Metric spaces have many applications in computer science—see e.g., [25, 34].

Having introduced metric spaces, we then show that the set of trees over some set can be viewed as a metric space. Early papers addressing this idea include [2, 3, 6, 7, 8, 9, 12, 27]. More recent references appear in [25, 34]. Lloyd [24] uses the metric space of trees to present complete Herbrand interpretations for non-terminating logic programs.

**Definition 18.** Let  $X$  be a set and  $d : X \times X \rightarrow \mathbb{R}_+$  a map<sup>9</sup> with, for all  $x, y, z \in X$ :

1.  $d(x, y) = d(y, x)$ ;
2.  $d(x, y) = 0$  iff  $x = y$ ;
3.  $d(x, y) + d(y, z) \geq d(x, z)$ .

Then  $d$  is a *metric* on  $X$ , and  $(X, d)$  is a *metric space*.

*Example 10.*

1. The function  $d(x, y) = |x - y|$  is a metric on  $\mathbb{R}$ .
2. For a set  $X$ , the map  $d : X \times X \rightarrow \mathbb{R}_+$ ,

$$d_X(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$

is a metric on  $X$ , called the *discrete metric* on  $X$ .

**Definition 19.** Let  $(X, d)$  be a metric space.

1. A sequence  $x_0, x_1, \dots \in X$  *stabilizes to*  $x \in X$  if there exists an  $N$  such that, for all  $n \geq N$ ,  $d(x_n, x) = 0$ .
2. A sequence  $x_0, x_1, \dots \in X$  is *convergent* with *limit*  $x \in X$  if, for all  $\varepsilon > 0$ , there exists an  $N$  such that, for all  $n \geq N$ ,  $d(x_n, x) \leq \varepsilon$ .
3. A sequence  $x_0, x_1, \dots \in X$  is a *Cauchy sequence* if, for all  $\varepsilon > 0$ , there exists an  $N$  such that, for all  $m, n \geq N$ ,  $d(x_n, x_m) \leq \varepsilon$ .

*Remark.* Let  $(X, d)$  be a metric space.

---

<sup>9</sup>  $\mathbb{R}_+ = \{r \in \mathbb{R} \mid r \geq 0\}$ .

1. A stabilizing sequence is convergent, and a convergent sequence is a Cauchy sequence. None of the converse implications hold in general.
2. Any sequence has at most one limit.

**Definition 20.** Let  $(X, d)$  be a metric space. If every Cauchy sequence in  $(X, d)$  is convergent then  $(X, d)$  is *complete*.

**Definition 21.** Let  $(X, d), (Y, d')$  be metric spaces. A map  $f : X \rightarrow Y$  is *continuous at*<sup>10</sup>  $x \in X$  if, for every sequence  $x_0, x_1, \dots \in X$  that converges to  $x$ ,  $f(x_0), f(x_1), \dots \in Y$  converges to  $f(x)$ . Also,  $f : X \rightarrow Y$  is *continuous* if  $f$  is continuous at every  $x \in X$ .

*Example 11.* Let  $(X, d)$  be a metric space. Let  $d_{\mathbb{B}}$  be the discrete metric on  $\mathbb{B} = \{0, 1\}$ . It is natural to view a predicate on  $X$  as a function  $p : X \rightarrow \mathbb{B}$ , and say that  $p(x)$  is true and false if  $p(x) = 1$  and  $p(x) = 0$ , respectively.

Then  $p$  is continuous iff for every sequence  $x_0, x_1, \dots$  that converges to  $x$ , the sequence  $p(x_0), p(x_1), \dots$  converges to  $p(x)$ .

*Remark.* Let  $(X, d_X), (Y, d_Y)$ , and  $(Z, d_Z)$  be metric spaces. If  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$  are both continuous, then so is  $g \circ f : X \rightarrow Z$ .

In the rest of this section  $E$  is some set. What is the distance between  $t, t' \in T_{\infty}(E)$ ? It is natural to require that trees which have large coinciding initial subtrees are close.

**Definition 22.** Define  $d : T_{\infty}(E) \times T_{\infty}(E) \rightarrow \mathbb{R}_+$  by:

$$d(t, t') = \begin{cases} 0 & \text{if } t = t' \\ 2^{-\min\{\ell \mid t[\ell] \neq t'[\ell]\}} & \text{otherwise} \end{cases}$$

It is a routine exercise to verify that  $(T_{\infty}(E), d)$  is indeed a metric space, which we call the *metric space of trees* (over  $E$ ).

*Remark.*

1. A sequence  $t_0, t_1, \dots \in T_{\infty}(E)$  stabilizes to  $t$  iff there exists an  $N$  such that, for all  $n \geq N$ ,  $t_n = t$ .
2. A sequence  $t_0, t_1, \dots \in T_{\infty}(E)$  converges to  $t$  iff for all  $\ell$ , there exists an  $N$  such that, for all  $n \geq N$ ,  $t_n[\ell] = t[\ell]$ .
3. A sequence  $t_0, t_1, \dots \in T_{\infty}(E)$  is a Cauchy sequence iff for all  $\ell$ , there exists an  $N$  such that, for all  $n \geq N$ ,  $t_n[\ell] = t_{n+1}[\ell]$ .

The next result was first proved by Bloom, Elgot and Wright [7], and independently noted by Mycielski and Taylor [27] and Arnold and Nivat [3, 2].

**Proposition 23.** *The metric space  $(T_{\infty}(E), d)$  is complete.*

The following connection between stability, convergence, and predicates does not hold in arbitrary metric spaces.

**Lemma 24.** A predicate  $p$  on  $T_{\infty}(E)$  is continuous iff for every convergent sequence  $t_0, t_1, \dots \in T_{\infty}(E)$  with *infinite* limit  $t$ , the sequence  $p(t_0), p(t_1), \dots$  stabilizes to  $p(t)$ .

<sup>10</sup> This is not the usual definition of continuity, but it is well-known that this definition is equivalent to the usual one.

## 6 Termination of Transformers

We now give a condition ensuring termination of an abstract program transformer.

The idea in ensuring termination of an apt is that it maintains some invariant. For instance, a transformer might never introduce a node whose label is larger, in some order, than the label on the parent node. In cases where an unfolding step would render the invariant false, some kind of generalization is performed.

**Definition 25.** Let  $M : T(E) \rightarrow T(E)$  be an apt on  $E$  and  $p : T_\infty(E) \rightarrow \mathbb{B}$  be a predicate.  $M$  *maintains*  $p$  if, for every singleton  $t \in T(E)$  and  $i \in \mathbb{N}$ ,  $p(M^i(t)) = 1$ .

**Definition 26.** A predicate  $p : T_\infty(E) \rightarrow \mathbb{B}$  is *finitary* if  $p(t) = 0$  for all infinite  $t \in T_\infty(E)$ .

**Definition 27.** An apt  $M$  on  $E$  is *Cauchy* if, for every singleton  $t \in T_\infty(E)$ , the sequence  $t, M(t), M^2(t), \dots$  is a Cauchy sequence.

The following theorem gives a sufficient condition for a program transformer to terminate.

**Theorem 28.** Let apt  $M : T(E) \rightarrow T(E)$  maintain predicate  $p : T_\infty(E) \rightarrow \mathbb{B}$ . If

1.  $M$  is Cauchy, and
2.  $p$  is finitary and continuous,

then  $M$  terminates.

Informally, the condition that  $M$  be Cauchy guarantees that *only finitely many generalization steps* will happen at a given node, and the condition that  $p$  be finitary and continuous guarantees that *only finitely many unfolding steps* will be used to expand the transformation tree. The first condition can be satisfied by adopting appropriate unfolding and generalization operations, and the second condition can be satisfied by adopting an appropriate criterion for deciding when to generalize.

Next we consider specific techniques for ensuring that an apt is Cauchy and that a predicate is finitary and continuous. We begin with the former.

**Definition 29.** Let  $S$  be a set with a relation  $\leq$ . Then  $(S, \leq)$  is a *quasi-order* if  $\leq$  is reflexive and transitive. We write  $s < s'$  if  $s \leq s'$  and  $s' \not\leq s$ .

**Definition 30.** Let  $(S, \leq)$  be a quasi-order.

1.  $(S, \leq)$  is *well-founded* if there is no infinite sequence  $s_0, s_1, \dots \in S$  with  $s_0 > s_1 > \dots$
2.  $(S, \leq)$  is a *well-quasi-order* if, for every infinite sequence  $s_0, s_1, \dots \in S$ , there are  $i < j$  with  $s_i \leq s_j$ .

An apt is Cauchy if it always either adds some new children to a leaf node (unfolds), or replaces a subtree by a new tree whose root label is strictly smaller than the label of the root of the former subtree (generalizes). This is how most online transformers work.



**Proposition 31.** *Let  $(E, \leq)$  be a well-founded quasi-order and  $M : T(E) \rightarrow T(E)$  an apt such that, for all  $t$ ,  $M(t) = t\{\gamma := t'\}$  for some  $\gamma, t'$  where*

1.  $\gamma \in \text{leaf}(t)$  and  $t(\gamma) = t'(\epsilon)$  (unfold); or
2.  $t(\gamma) > t'(\epsilon)$  (generalize).

*Then  $M$  is Cauchy.*

Next we consider ways of ensuring that a predicate is (finitary and) continuous.

A family  $S$  of sets is of *finite character* if each set is a member if and only if all its finite subsets are members. Adapted to families of *trees*, the definition reads:

**Definition 32.** A family  $T \subseteq T_\infty(E)$  of trees is of *finite character* iff, for all  $t \in T_\infty(E)$ :

$$t \in T \Leftrightarrow \forall \ell \in \mathbb{N} : t[\ell] \in T$$

The following shows that a finitary predicate  $p : T_\infty(E) \rightarrow \mathbb{B}$  is continuous, if the family  $\{t \mid p(t) = 1\}$  is of finite character.

**Proposition 33.** *Suppose  $p : T_\infty(E) \rightarrow \mathbb{B}$  is finitary and, for all  $t \in T_\infty(E)$ ,*

$$p(t) = 1 \Leftrightarrow \forall \ell \in \mathbb{N} : p(t[\ell]) = 1$$

*Then  $p$  is continuous.*

We end the section by reviewing instances of Proposition 33.

The following shows that a Cauchy transformer terminates if it never introduces a node whose label is larger than an ancestor's label with respect to some well-quasi-order. This idea is used in a number of transformers [1, 16, 23, 35, 41]

**Proposition 34.** *Let  $(E, \leq)$  be a well-quasi-order. Then  $p : T_\infty(E) \rightarrow \mathbb{B}$ ,*

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \alpha i \beta \in \text{dom}(t) : t(\alpha) \leq t(\alpha i \beta) \\ 1 & \text{otherwise} \end{cases}$$

*is finitary and continuous.*

The following shows that a Cauchy transformer terminates if it never introduces a node whose label is not smaller than its *immediate* ancestor's label with respect to some well-founded quasi-order.

**Proposition 35.** *Let  $(E, \leq)$  be a well-founded quasi-order. Then  $p : T_\infty(E) \rightarrow \mathbb{B}$ ,*

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \alpha i \in \text{dom}(t) : t(\alpha) \not\geq t(\alpha i) \\ 1 & \text{otherwise} \end{cases}$$

*is finitary and continuous.*

The following generalization of the preceding proposition is used in some techniques for ensuring global termination of partial deduction [26].

**Proposition 36.** Let  $\{E_1, \dots, E_n\}$  be a partition<sup>11</sup> of  $E$  and  $\leq_1, \dots, \leq_n$  be well-founded quasi-orders on  $E_1, \dots, E_n$ , respectively. Then  $p: T_\infty(E) \rightarrow \mathbb{B}$ ,

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \alpha i \beta \in \text{dom}(t), j \in \{1, \dots, n\} : t(\alpha), t(\alpha i \beta) \in E_j \text{ \& } t(\alpha) \not\preceq_j t(\alpha i \beta) \\ 1 & \text{otherwise} \end{cases}$$

is finitary and continuous.

The following shows that one can combine well-quasi-orders and well-founded quasi-orders in a partition.

**Proposition 37.** Let  $\{E_1, E_2\}$  be a partition of  $E$  and let  $\leq_1$  be a well-quasi-order on  $E_1$  and  $\leq_2$  a well-founded quasi-order on  $E_2$ . Then  $p: T_\infty(E) \rightarrow \mathbb{B}$ ,

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \alpha i \beta \in \text{dom}(t) : t(\alpha), t(\alpha i \beta) \in E_1 \text{ \& } t(\alpha) \not\leq_1 t(\alpha i \beta) \\ 0 & \text{if } \exists \alpha, \alpha i \in \text{dom}(t) : t(\alpha), t(\alpha i) \in E_2 \text{ \& } t(\alpha) \not\preceq_2 t(\alpha i) \\ 1 & \text{otherwise} \end{cases}$$

is finitary and continuous.

The following shows that it suffices to apply a finitary and continuous predicate to the *interior* part of a tree.

**Definition 38.** For  $t \in T_\infty(E)$ , define the *interior*  $t^0 \in T_\infty(E)$  of  $t$  by:

$$\begin{aligned} \text{dom}(t^0) &= (\text{dom}(t) \setminus \text{leaf}(t)) \cup \{\epsilon\} \\ t^0(\gamma) &= t(\gamma) \quad \text{for all } \gamma \in \text{dom}(t^0) \end{aligned}$$

**Proposition 39.** Let  $p: T_\infty(E) \rightarrow \mathbb{B}$  be finitary and continuous. Then also the map  $q: T_\infty(E) \rightarrow \mathbb{B}$  defined by

$$q(t) = p(t^0)$$

is finitary and continuous.

It is not hard to see that one can replace in the proposition  $\bullet^0$  by any continuous map which maps infinite trees to infinite trees.

## 7 Application: Termination of Positive Supercompilation

In this section we prove that positive supercompilation  $M_{ps}$  terminates. We do so by proving that  $M_{ps}$  is Cauchy and that  $M_{ps}$  maintains a finitary, continuous predicate.

We first prove that  $M_{ps}$  is Cauchy; the idea is to use Proposition 31. Indeed,  $M_{ps}$  always either unfolds by a driving step or replaces a subtree by a new leaf whose label is strictly smaller than the expression in the root of the former subtree. In which order?

**Proposition 40.**  $M_{ps}$  is Cauchy.

<sup>11</sup> That is,  $E_1, \dots, E_n$  are sets with  $\cup_{i=1}^n E_i = E$  and  $i \neq j \Rightarrow E_i \cap E_j = \emptyset$ .

*Proof.* Define the relation  $\succ$  on  $\mathcal{L}$  by:

$$\mathbf{let } x'_1=e'_1, \dots, x'_m=e'_m \mathbf{ in } e \succ \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } e \Leftrightarrow m = 0 \ \& \ n \geq 0$$

It is a routine exercise to verify that  $\succeq$  is a well-founded quasi-order.

We now show that for any  $t \in T(\mathcal{L})$

$$M_{ps}(t) = t\{\gamma := t'\}$$

where, for some  $\gamma \in \text{dom}(t)$  and  $t' \in T_\infty(\mathcal{L})$ , either  $\gamma \in \text{leaf}(t)$  and  $t(\gamma) = t'(\epsilon)$ , or  $t(\gamma) \succ t'(\epsilon)$ . We proceed by case analysis of the operation performed by  $M_{ps}$ .

1.  $M_{ps}(t) = \text{drive}(t, \gamma) = t\{\gamma := t'\}$ , where  $\gamma \in \text{leaf}(t)$  and, for certain expressions  $e_1, \dots, e_n$ ,  $t' = t(\gamma) \rightarrow e_1, \dots, e_n$ . Then

$$t(\gamma) = t'(\epsilon)$$

2.  $M_{ps}(t) = \text{abstract}(t, \gamma, \alpha) = t\{\gamma := \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } e \rightarrow\}$ , where  $\alpha \in \text{anc}(t, \gamma)$ ,  $t(\alpha) \neq t(\gamma)$ ,  $t(\alpha), t(\gamma) \in \mathcal{E}$  are both non-trivial,  $t(\alpha) \leq t(\gamma)$ ,  $e = t(\alpha) \sqcap t(\gamma)$ , and  $t(\gamma) = e\{x_1:=e_1, \dots, x_n:=e_n\}$ . Then  $e = t(\alpha)$  and  $t(\gamma) = t(\alpha)\{x_1:=e_1, \dots, x_n:=e_n\}$ , but  $t(\gamma) \neq t(\alpha)$ , so  $n > 0$ . Thus :

$$t(\gamma) \succ \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } e = t'(\epsilon)$$

3.  $M_{ps}(t) = \text{abstract}(t, \gamma, \beta) = t\{\gamma := \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } e \rightarrow\}$ , where  $\gamma \in \text{anc}(t, \beta)$ ,  $t(\beta), t(\gamma)$  are both non-trivial,  $t(\gamma) \not\leq t(\beta)$ ,  $e = t(\gamma) \sqcap t(\beta)$ , and where we also have  $t(\gamma) = e\{x_1:=e_1, \dots, x_n:=e_n\}$ . Then  $t(\gamma) \neq e$ , but  $t(\gamma) = e\{x_1:=e_1, \dots, x_n:=e_n\}$ , so  $n > 0$ . Thus:

$$t(\gamma) \succ \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } e = t'(\epsilon)$$

4.  $M_{ps}(t) = \text{split}(t, \gamma) = t\{\gamma := \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } h(x_1, \dots, x_n) \rightarrow\}$  where, for some  $\alpha \in \text{anc}(t, \gamma)$ ,  $t(\alpha), t(\beta)$  are non-trivial,  $t(\alpha) \leq^* t(\gamma)$ ,  $t(\alpha) \leftrightarrow t(\gamma)$ , and also  $t(\gamma) = h(e_1, \dots, e_n)$ , where  $h \in C \cup F \cup G$ .<sup>12</sup> Here  $n > 0$ : if  $n = 0$ , then  $t(\gamma) = h()$ , but then  $t(\alpha) \not\leftrightarrow t(\beta)$ . Thus,

$$t(\gamma) = h(e_1, \dots, e_n) \succ \mathbf{let } x_1=e_1, \dots, x_n=e_n \mathbf{ in } h(x_1, \dots, x_n) = t'(\epsilon)$$

This concludes the proof.  $\square$

Next we prove that  $M_{ps}$  maintains a finitary, continuous predicate.

The following result, known as *Kruskal's Tree Theorem*, is due to Higman [19] and Kruskal [22]. Its classical proof is due to Nash-Williams [28].

**Theorem 41.**  $(\mathcal{E}_H(V), \leq)$  is a well-quasi-order, provided  $H$  is finite.

*Proof.* Collapse all variables to one 0-ary operator and use the proof in [14].  $\square$

**Corollary 42.** The relation  $\leq^*$  is a well-quasi order on  $\mathcal{E}$ .

<sup>12</sup> Since  $t(\gamma)$  is non-trivial,  $t(\gamma)$  must have form  $h(e_1, \dots, e_n)$ .

*Proof.* Given an infinite sequence  $e_0, e_1, \dots \in \mathcal{E}$  there must be an infinite subsequence  $e_{i_0}, e_{i_1}, \dots$  such that  $B(e_{i_0}) = B(e_{i_1}) = \dots$ . By Theorem 41,<sup>13</sup> there are  $k$  and  $l$  such that  $e_{i_k} \sqsubseteq e_{i_l}$  and then  $e_{i_k} \sqsubseteq^* e_{i_l}$ , as required.  $\square$

**Proposition 43.**  $M_{ps}$  maintains a finitary, continuous predicate.

*Proof.* Define  $|\bullet| : \mathcal{E} \rightarrow \mathbb{N}$  by

$$\begin{aligned} |g(e_0, e_1, \dots, e_m)| &= 1 + |e_0| + \dots + |e_m| \\ |f(e_1, \dots, e_m)| &= 1 + |e_1| + \dots + |e_m| \\ |c(e_1, \dots, e_m)| &= 1 + |e_1| + \dots + |e_m| \\ |x| &= 1 \end{aligned}$$

Define  $l : \mathcal{L} \rightarrow \mathcal{E}$  by:

$$l(\text{let } x_1=e_1, \dots, x_n=e_n \text{ in } e) = e\{x_1 := e_1, \dots, x_n := e_n\}$$

for  $n \geq 0$ .

Finally, define  $\sqsubseteq$  on  $\mathcal{L}$  by:

$$\ell \sqsubseteq \ell' \quad \Leftrightarrow \quad |l(\ell)| > |l(\ell')| \text{ or } |l(\ell)| = |l(\ell')| \ \& \ l(\ell) \geq l(\ell')$$

It is a routine exercise to verify that  $\sqsubseteq$  is a well-founded quasi-order using the fact that  $\sqsubseteq$  is well-founded.

Consider the predicate  $q : T_\infty(\mathcal{L}) \rightarrow \mathbb{B}$  defined by

$$q(t) = p(t^0)$$

where  $p : T_\infty(\mathcal{L}) \rightarrow \mathbb{B}$  is defined by:

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \alpha i \beta \in \text{dom}(t) : t(\alpha), t(\alpha i \beta) \text{ are non-trivial} \ \& \ t(\alpha) \sqsubseteq^* t(\alpha i \beta) \\ 0 & \text{if } \exists \alpha, \alpha i \in \text{dom}(t) : t(\alpha), t(\alpha i) \text{ are trivial} \ \& \ t(\alpha) \not\sqsubseteq^* t(\alpha i) \\ 1 & \text{otherwise} \end{cases}$$

The sets of non-trivial and trivial expressions constitute a partition of  $\mathcal{L}$ . Also,  $\sqsubseteq^*$  is a well-quasi-order on the set of non-trivial expressions (in fact, on all of  $\mathcal{E}$ ) and  $\sqsubseteq$  is a well-founded quasi-order on the set of trivial expressions (in fact, on all of  $\mathcal{L}$ ). It follows by Proposition 37 that  $p$  is finitary and continuous, and then by Proposition 39 that  $q$  is also finitary and continuous.

It remains to show that  $M_{ps}$  maintains  $q$ , i.e., that  $q(M_{ps}^i(t_0)) = 1$  for any singleton  $t_0 \in T_\infty(\mathcal{L})$ .

Given any  $t \in T_\infty(\mathcal{L})$  and  $\beta \in \text{dom}(t)$ , we say that  $\beta$  is *good* in  $t$  if the following conditions both hold:

- (i)  $t(\beta)$  non-trivial  $\& \ \beta \notin \text{leaf}(t) \Rightarrow \forall \alpha \in \text{anc}(t, \beta) \setminus \{\beta\} : t(\alpha) \text{ non-trivial} \Rightarrow t(\alpha) \not\sqsubseteq^* t(\beta)$ ;
- (ii)  $\beta = \alpha i \ \& \ t(\alpha) \text{ trivial} \Rightarrow t(\alpha) \sqsupset t(\beta)$ .

<sup>13</sup> Recall that  $\mathcal{E} = \mathcal{E}_{F \cup G \cup C}$  where  $F, G, C$  are finite.

We say that  $t$  is *good* if all  $\beta \in \text{dom}(t)$  are good in  $t$ .

It is easy to see that  $q(t) = 1$  if  $t$  is good (the converse does not hold). It therefore suffices to show for any singleton  $t_0 \in T_\infty(\mathcal{L})$  that  $M_{ps}^i(t_0)$  is good for all  $i$ . We proceed by induction on  $i$ .

For  $i = 0$ , (i)-(ii) are both vacuously satisfied since  $t_0$  consists of a single leaf.

For  $i > 0$ , we split into cases according to the operation performed by  $M_{ps}$  on  $M_{ps}^{i-1}(t_0)$ . Before considering these cases, note that by the definition of goodness, if  $t \in T_\infty(\mathcal{L})$  is good,  $\gamma \in \text{dom}(t)$ , and  $t' \in T_\infty(\mathcal{L})$ , then  $t\{\gamma := t'\}$  is good too, provided  $\gamma\delta$  is good in  $t\{\gamma := t'\}$  for all  $\delta \in \text{dom}(t')$ .

For brevity, let  $t = M_{ps}^{i-1}(t_0)$ .

1.  $M_{ps}(t) = \text{drive}(t, \gamma) = t\{\gamma := t'\}$ , where  $\gamma \in \text{leaf}(t)$ ,  $t' = t(\gamma) \rightarrow e_1, \dots, e_n$ , and  $\{e_1, \dots, e_n\} = \{e \mid t(\gamma) \Rightarrow e\}$ .

We must show that  $\gamma, \gamma 1, \dots, \gamma n$  are good in  $M_{ps}(t)$ .

To see that  $\gamma$  is good in  $M_{ps}(t)$ , note that if  $t(\gamma)$  is non-trivial, then the algorithm ensures that condition (i) is satisfied. Condition (ii) follows from the induction hypothesis.

To see that  $\gamma i$  is good in  $M_{ps}(t)$ , note that condition (i) is vacuously satisfied. Moreover, when  $\ell \Rightarrow e$  and  $\ell$  is trivial,  $\ell \sqsupset e$ , so condition (ii) holds as well.

2.  $M_{ps}(t) = \text{abstract}(t, \gamma, \alpha) = t\{\gamma := \text{let } x_1 = e_1, \dots, x_n = e_n \text{ in } e \rightarrow\}$ , where  $\alpha \in \text{anc}(t, \gamma)$ ,  $t(\alpha) \neq t(\gamma)$ ,  $t(\alpha), t(\gamma) \in \mathcal{E}$  are both non-trivial,  $t(\alpha) \leq t(\gamma)$ ,  $e = t(\alpha) \sqcap t(\gamma)$ , and  $t(\gamma) = e\{x_1 := e_1, \dots, x_n := e_n\}$ .

We must show that  $\gamma$  is good in  $M_{ps}(t)$ . Condition (i) holds vacuously, and (ii) follows from the induction hypothesis and  $l(t(\gamma)) = l(\text{let } x_1 = e_1, \dots, x_n = e_n \text{ in } e)$ .

The remaining two cases are similar to the preceding case. □

Martens and Gallagher show, essentially, that an abstract program transformer terminates if it maintains a predicate of the form in Proposition 36 and always either adds children to a node or replaces a subtree with root label  $e$  by a new node whose label  $e'$  is in the same partition  $E_j$  as  $e$  and  $e >_j e'$ . In our setting this result follows from Propositions 31 and 36 (by Theorem 28).

Martens and Gallagher then go on to show that a certain generic partial deduction algorithm always terminates; this result follows from the above more general result. For brevity we omit the details.

*Acknowledgments.* This work grew out of joint work with Robert Glück. I am indebted to Nils Andersen and Klaus Grue for discussions about metric spaces. Thanks to Maria Alpuente, Nils Andersen, Robert Glück, Laura Lafave, Michael Leuschel, Bern Martens, and Jens Peter Secher for comments to an early version of this paper.

## References

1. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H.R. Nielson, editor, *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 1996.
2. A. Arnold and M. Nivat. Metric interpretations of infinite trees and semantics of non deterministic recursive programs. *Theoretical Computer Science*, 11:181–205, 1980.

3. A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, III(4):445–476, 1980.
4. R. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.
5. D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.
6. S.L. Bloom. All solutions of a system of recursion equations in infinite trees and other contraction theories. *Journal of Computer and System Sciences*, 27:225–255, 1983.
7. S.L. Bloom, C.C. Elgot, and J.B. Wright. Vector iteration in pointed iterative theories. *SIAM Journal of Computing*, 9(3):525–540, 1980.
8. S.L. Bloom and D. Patterson. Easy solutions are hard to find. In *Colloquium on Trees in Algebra and Programming*, volume 112 of *Lecture Notes in Computer Science*, pages 135–146. Springer-Verlag, 1981.
9. S.L. Bloom and R. Tindell. Compatible orderings on the metric theory of trees. *SIAM Journal of Computing*, 9(4):683–691, 1980.
10. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic programming*, 6:135–162, 1989.
11. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machines*, 24(1):44–67, 1977.
12. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
13. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
14. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3, 1987.
15. Y. Futamura and K. Nogi. Generalized partial computation. In Bjørner et al. [5], pages 133–151.
16. 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.
17. 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.
18. R. Glück and M.H. Sørensen. A roadmap to metacomputation by supercompilation. In Danvy et al. [13], pages 137–160.
19. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, (3) 2:326–336, 1952.
20. 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.
21. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
22. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
23. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Danvy et al. [13], pages 263–283.
24. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

25. M. Main, A. Melton, M. Mislove, and D. Schmidt, editors. *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.
26. 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.
27. J. Mycielski and W. Taylor. A compactification of the algebra of terms. *Algebra Universalis*, 6:159–163, 1976.
28. C.St.J.A. Nash-Williams. On well-quasi-ordering finite trees. *Proceedings of the Cambridge Mathematical Society*, 59:833–835, 1963.
29. A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Conference on Lisp and Functional Programming*, pages 273–281. ACM Press, 1984.
30. A. Pettorossi and M. Proietti. A comparative revisitation of some program transformation techniques. In Danvy et al. [13], pages 355–385.
31. M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic programming*, 16:123–161, 1993.
32. W. Rudin. *Principles of Mathematical Analysis*. Mathematics Series. McGraw-Hill, third edition, 1976.
33. E. Ruf and D. Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, 1993.
34. M.B. Smyth. Topology. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 641–761. Oxford University Press, 1992.
35. 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.
36. M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1994.
37. M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
38. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *International Conference on Logic Programming*, pages 127–138. Uppsala University, 1984.
39. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
40. V.F. Turchin. The algorithm of generalization in the supercompiler. In Bjørner et al. [5], pages 531–549.
41. V.F. Turchin. On generalization of lists and strings in supercompilation. Technical Report CSc. TR 96-002, City College of the City University of New York, 1996.
42. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.
43. D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191. Springer-Verlag, 1991.