ELSEVIER

Science of Computer Programming 37 (2000) 163–205

Science of Computer Programming

www.elsevier.nl/locate/scico

Morten Heine B. Sørensen

Department of Computer Science, University of Copenhagen (DIKU), Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

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 that use 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. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Program transformers; Supercompilation; Termination; Metric space of trees; Generalization

1. Introduction

Numerous program transformation techniques have been studied in the areas of functional and logic languages, e.g. partial evaluation (see [5,13]) and unfold/fold transformations. Pettorossi and Proietti [33] show that many of these techniques can be

☆ This is an elaborated version of a conference paper [41] with the same title. The main additions comprise: a number of examples; all the proofs in Sections 5 and 6; and the application to two new variants of positive supercompilation in Section 7.

E-mail address: rambo@diku.dk (M.H.B. Sørensen).

PII: S0167-6423(99)00026-X

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 [38], have been adopted in partial deduction [26], conjunctive partial deduction [19], Turchin's supercompiler [45], 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 that 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 that use 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 only 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 variations of positive supercompilation as abstract program transformers. 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 the different variations of positive supercompilation terminate. The section also gives a sketch of how Martens and Gallagher's [29] generic algorithm for partial deduction can be viewed as an abstract program transformer and of a proof 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 [20, 21, 23, 39, 40] and to develop a unifying theory for such transformers.

2. Trees in transformation

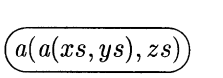
We now proceed to show how program transformers may be viewed as maps that manipulate certain trees, following Pettorossi and Proietti [33].

Example 1. Consider a function program appending two lists.

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

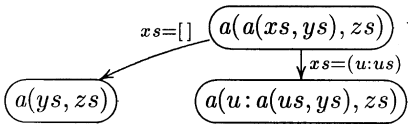
A simple and elegant way to append *three* lists is to use the expression $a(a(xs, ys), zs)$. However, this expression is inefficient since it traverses xs twice. We now illustrate a standard transformation obtaining a more efficient method.

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



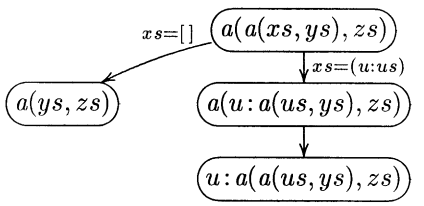
By an *unfolding* step which replaces the inner call to append according to the different patterns in the definition of a, two new expressions are added as labels on

children:

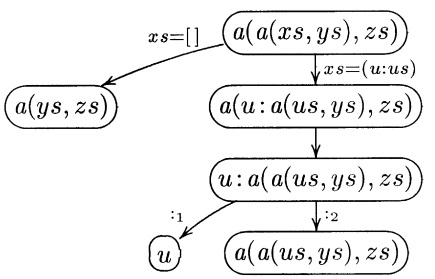


Unfolding steps are similar to evaluation steps in a small-step call-by-name operational semantics except that the former apply to expressions with variables.

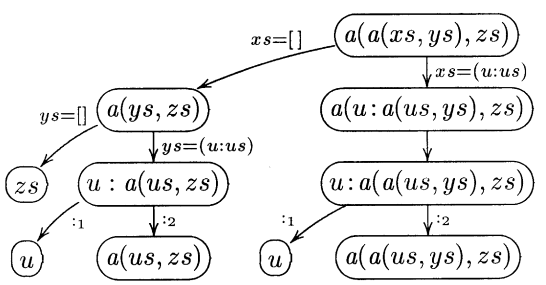
In the rightmost child we can perform an unfolding step, which replaces the outer call to append:



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 branches in the tree correspond to different run-time values for the free variables of e. In the above tree, computation starts in the root with values for xs, ys and zs, 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 a finite, closed tree, we proceed as follows.[1] For each node that has several children according to different patterns, we introduce a new function definition, where the left-hand side is derived from the node's label, and the right-hand sides are derived from the children. More specifically the left-hand side is obtained by renaming the label of the node to $f(x_1, \ldots, x_n)$, where f is a fresh function name and $x_1, \ldots, x_n$ are the distinct variables that occur in the term. The right-hand sides are obtained similarly from the children. The new term, which is to replace the original term that was transformed, is obtained similarly from the root node.

In the above example we rename expressions of form $a(a(xs, ys), zs)$ as $aa(xs, ys, zs)$, and we rename expressions of form $a(ys, zs)$ as $a'(ys, zs)$, and derive from the tree the following new program:[2]

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

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

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.

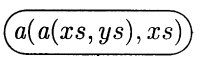
The transformation in Example 1 proceeded in three phases – symbolic computation, search for regularities, and program extraction – the first two of which were interleaved. In the first phase we performed unfolding steps that added children to the tree. In the second phase we made sure that no node with an expression which was a renaming of an 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.

[1] There are a number of exceptions to the following rules. In this paper we are concerned with techniques for automatically constructing transformation trees such as those above, in particular with how one can make the process terminate with a finite, closed tree. Generating programs from such trees is not difficult, but a few technicalities must be observed. Since code generation is not the topic of this paper, we omit a detailed account.

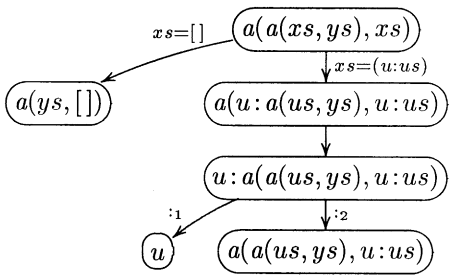
[2] Incidentally, the new function a' turns out to be a copy of the old function a.

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 following 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:

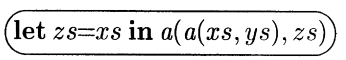


After the first few steps we have:



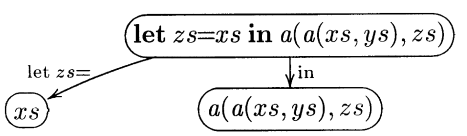
Unlike the situation in Example 1, the label of the rightmost node is not a renaming of the expression at the root. In fact, repeated unfolding will *never* lead to that situation; special measures must be taken.

One solution is to remove the information that sets the two expressions apart – the second argument in the outer call to append. This is achieved by a *generalization* step that replaces the whole tree by a single new node:



Another way of perceiving this step is that we have extracted the common structure $a(a(xs, ys), zs)$ of the two expressions (inserting fresh variables in the places where the structure of the two expressions differ).

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.[3]

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 [26, 29], conjunctive partial deduction [19], compiling control [10], loop absorption [34], partial evaluation of functional logic languages [1], unfold/fold transformation of functional programs [11], unfold/fold transformation of logic programs [42], tupling [4, 32], supercompilation [43, 44], positive supercompilation [21, 38], generalized partial computation [18], deforestation [46], and online partial evaluation of functional programs [24, 36, 47].

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.

Most of the following definition is taken from Courcelle [12].

Definition 1. A *tree* over a set E is a partial map[4] $t : \mathbb{N}_1^* \to E$ 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 $1 \leqslant i \leqslant j$ (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 *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 leafs 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 $\mathrm{dom}(t) = \{\varepsilon\}$, i.e. if $\mathrm{dom}(t)$ 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.

[3] 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.

[4] We let $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$. S^* is the set of finite strings over S. We use $i, j, k \in \mathbb{N}_1$ and $\alpha, \beta, \gamma \in \mathbb{N}_1^*$. Finally, $\mathrm{dom}(f)$ is the domain of a partial function f.

Example 3. Let $\mathscr{E}_H(V)$ 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. Then let $\mathscr{L}_H(V)$ be the smallest set such that $e_1, \ldots, e_n,\ e \in \mathscr{E}_H(V)$ implies that **let** $x_1 = e_1, \ldots, x_n = e_n$ **in** $e \in \mathscr{L}_H(V)$. The trees in Example 1 and 2 (ignoring labels on edges) are a diagrammatical presentation of trees over $\mathscr{E}_H(V)$ and $\mathscr{L}_H(V)$, respectively.

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

An apt only computes a *single* step of transformation: it maps some tree to a new tree by performing, e.g. an unfolding step. Hence, the sequences of trees in Examples 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 M 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 $i \in \mathbb{N}$.[5]
2. An apt M on E *terminates* if M terminates on all singletons $t \in T(E)$.

Although apts 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 α in t is:

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

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

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

3. For $\ell \in \mathbb{N}$, the *initial subtree of depth* ℓ *of* t, written $t[\ell]$, is the tree t' with

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

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

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

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

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

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

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

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

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

In the diagrammatical notation of Section 2, the depth of a node is the number of edges in the branch from the root to the node. The depth of a tree is the maximal depth of any node. The initial subtree of depth ℓ is the tree obtained by deleting all nodes of depth greater than ℓ and edges into such nodes. 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 branch from the root to the node, excluding the node itself. Finally, the tree $e \to e_1, \ldots, e_n$ is the tree with root labeled e and n children labeled $e_1, \ldots, e_n$, respectively.

4. Example: positive supercompilation

In this section, we present three variants of positive supercompilation [21, 38–41] as abstract program transformers.

The first subsection introduces the language for which we shall state the positive supercompilers. The second subsection presents the unfolding operations used in positive supercompilation. The two next subsections introduce the generalization operations, covering *when* to generalize and *how* to generalize, respectively. The last three subsections then introduce three variants of positive supercompilation which differ in when generalization is performed.

4.1. Language

We consider the following first-order functional language from [17]; 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 $\mathscr{Q}$ of programs, $\mathscr{D}$ of definitions, $\mathscr{E}$ of expressions, and $\mathscr{P}$ of patterns are defined by

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

where $m > 0$, $n \geqslant 0$. 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 vars(e) we denote the set of variables occurring in the expression e.

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

Remark. There is a close relationship between the set $\mathscr{E}$ of expressions introduced above and the set $\mathscr{E}_H(V)$ introduced in Example 3. In fact, $\mathscr{E} = \mathscr{E}_{C \cup F \cup G}(X)$. Therefore, in what follows we can make use of well-known results and definitions for $\mathscr{E}_H(V)$ in reasoning about $\mathscr{E}$. For instance, substitution on elements of $\mathscr{E}$ will be defined indirectly, by defining substitution on elements of $\mathscr{E}_H(V)$.

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 $\mathscr{L}$, as defined below.

Definition 6. The set $\mathscr{L}$ of let-expressions is defined as follows:

$$\mathscr{L} \ni \ell ::= \textbf{let } x_1 = e_1,\ldots,x_n = e_n \textbf{ in } e$$

where $n \geqslant 0$. If $n = 0$ then we identify the expression **let** $x_1 = e_1,\ldots,x_n = e_n$ **in** e with e. Thus, $\mathscr{E}$ is a subset of $\mathscr{L}$. If $n > 0$, we call ℓ a *proper* let-expression.

In the last three subsections we state our positive supercompilers as maps M: $T(\mathscr{L}) \to T(\mathscr{L})$. Given a program $q \in \mathscr{Q}$ and an expression $e \in \mathscr{E}$, we can view e as

a member of $\mathscr{L}$. By iterated application of M to the singleton tree labeled by e we eventually get a finite tree, which is closed in a certain sense, from which a new program and term can be reconstructed. Since we are concerned only with termination of the process, reconstruction of the new program and term will not be considered.

4.2. Unfolding

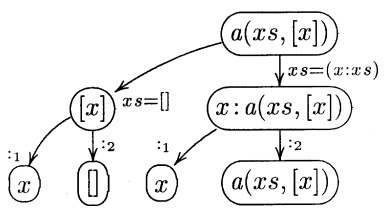
We now set out to formulate the unfolding operations used in positive supercompilation.

When we perform unfolding 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.

Example 5. Consider again the append program:

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

Suppose we wish to transform the term $a(xs,[x])$ that appends an element to the end of a list. We might construct the tree:



From this tree we can construct the new term $l(xs,x)$ and the program

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

In fact, this is not a program: the left-hand side with two occurrences of x is illegal.

How did the problem arise? We started out with the root expression $a(xs,[x])$ and instantiated xs to $(x:xs)$ arriving at a node labeled $x:a(xs,[x])$ in which the first x refers to the x in the pattern, and the second x refers to the x in the original term. We have confused different variables by giving them the same name x.

How can the problem be avoided? When instantiating a variable to a pattern we are free to use whatever variable names in the pattern we like, as long as we use the same names in the corresponding right-hand side of the function definition. Instead of $(x:xs)$ we should have taken a pattern with fresh names, e.g. $(u:us)$.

The following definitions introduce terminology to express freshness.

Definition 7. A *substitution* on $\mathscr{E}_H(V)$ is a total map from V to $\mathscr{E}_H(V)$. We denote by $\{x_1:=e_1,\ldots,x_n:=e_n\}$ the substitution that maps x_i to e_i (which we require to

be different from x_i) and all other variables to themselves. Substitutions are lifted to expressions as usual, and application of substitutions is written postfix.

Definition 8. Let θ be a substitution on $\mathscr{E}_H(V)$.

1. The *support* of θ is:

$$\text{support}(\theta) = \{x \in V \mid x\theta \neq x\};$$

2. The *yield* of θ is:

$$\text{yield}(\theta) = \bigcup \{\text{vars}(x\theta) \mid x \in \text{support}(\theta)\};$$

3. θ is *free for* $e \in \mathscr{E}_H(V)$ if

$$\text{yield}(\theta) \cap \text{vars}(e) = \emptyset.$$

The crucial property of a substitution θ which is free for an expression e is that the variables in the range of θ (at least those variables that are not simply mapped to themselves) do not occur already in e. In the above example we should have chosen the substitution that instantiates a variable to a pattern free for the expression containing the variable to be instantiated.

Unfolding steps add children to leaf nodes. The essence in defining the unfolding step is to define how the expressions in the new children are computed from the leaf's expression.

This computation is formalized by the following relation $\Rightarrow$ which is similar to the usual small-step semantics for normal-order reduction to weak head normal form. The relation extends the usual semantics 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 9. For a program q, the relations $e \rightarrow_\theta e'$ and $\ell \Rightarrow e$ where $e, e' \in \mathscr{E}$, $\ell \in \mathscr{L}$, and θ is a substitution on $\mathscr{E}$, are defined as in Fig. 1.

Example 6. Rules (1)–(3) are the base cases. For instance,

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

Rule (4) allows reduction in contexts, i.e. inside the first argument of a g-function. For instance,

$$a(a(xs, ys), xs) \rightarrow_{\{xs:=(u:us)\}} a(u : a(us, ys), xs) \quad \text{by Rule (4)}$$

Rules (5)–(7) are the main rules. For instance,

$$\begin{aligned} a(a(xs, ys), xs) &\Rightarrow a(u : a(us, ys), u : us) && \text{by Rule (5)} \\ u : a(a(us, ys), u : us) &\Rightarrow a(a(us, ys), u : us) && \text{by Rule (6)} \end{aligned}$$

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

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

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

$$\frac{e \rightarrow_\theta e'}{g(e,e_1,\ldots,e_n) \rightarrow_\theta g(e',e_1,\ldots,e_n)} \quad (4)$$

$$\frac{e \rightarrow_\theta e' \quad \& \quad \theta \text{ is free for } e}{e \Rightarrow e'\theta} \quad (5)$$

$$\frac{i \in \{1,\ldots,n\}}{c(e_1,\ldots,e_n) \Rightarrow e_i} \quad (6)$$

$$\frac{i \in \{1,\ldots,n+1\}}{\textbf{let } x_1{=}e_1,\ldots,x_n{=}e_n \textbf{ in } e_{n+1} \Rightarrow e_i} \quad (7)$$

Fig. 1. Generalized normal-order reduction.

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

Definition 10. Let $t \in T(\mathscr{L})$ and $\beta \in \text{leaf}(t)$. Then

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

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

The driving operation is illustrated in Fig. 2 together with some generalization operations introduced in a later subsection.

Example 7. All the unfolding steps in Examples 1–2 are, in fact, driving steps.

4.3. Generalization: when

Next we set out to formulate the generalization operations used in positive supercompilation. In this subsection we present the technique which decides *when* to generalize. The next subsection presents the actual generalization operations.

The following relation $\trianglelefteq$, adoped from [15], is used to decide when to generalize.

Definition 11. The *homeomorphic embedding* $\trianglelefteq$ is the smallest relation on $\mathscr{E}_H(V)$ such that, for all $h \in H$, $x, y, \in V$, and $e_i, e_i' \in \mathscr{E}_H(V)$:

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

where $m > 0$ and $n \geqslant 0$.

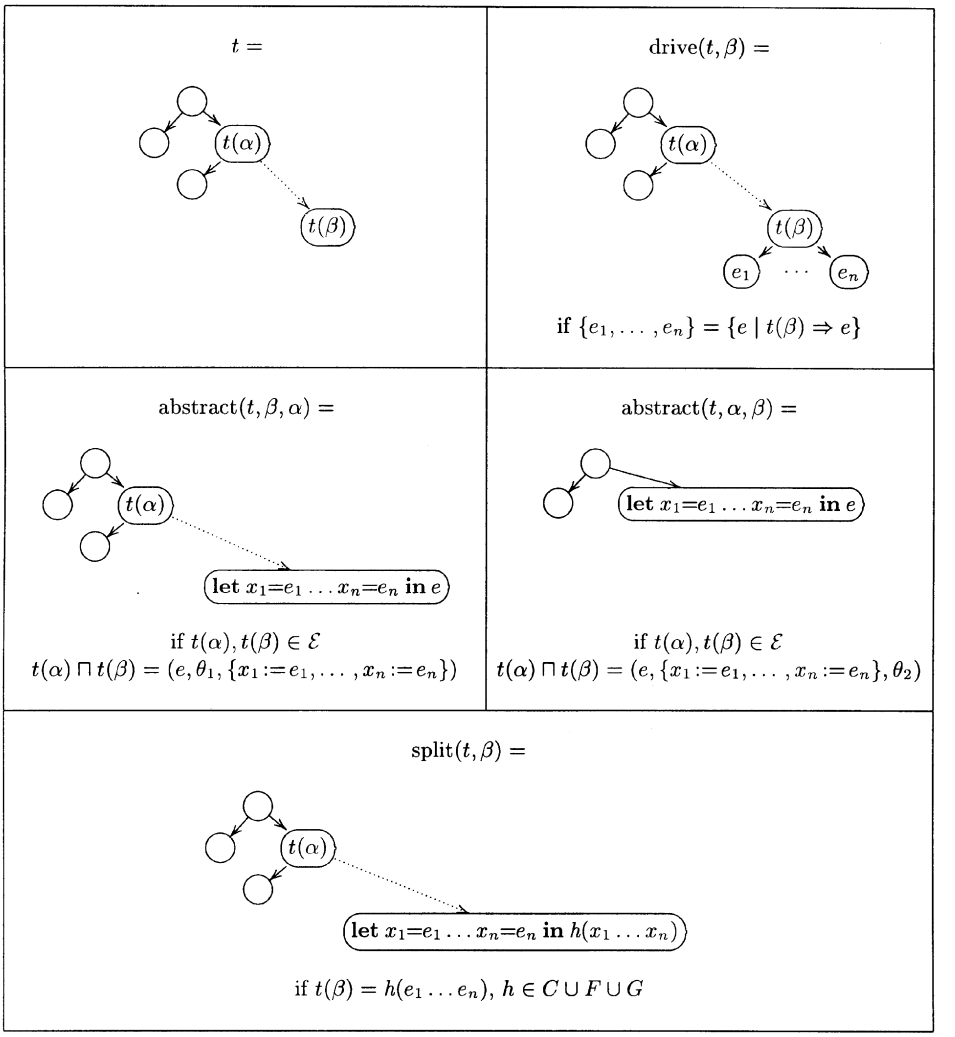


Fig. 2. Operations used in positive supercompilation.

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

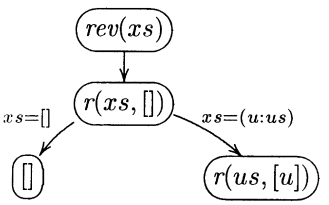
$$\begin{array}{ll} b \trianglelefteq f(b) & f(c(b)) \not\trianglelefteq c(b) \\ c(b) \trianglelefteq c(f(b)) & f(c(b)) \not\trianglelefteq c(f(b)) \\ c(b,b) \trianglelefteq c(f(b), f(b)) & f(c(b)) \not\trianglelefteq f(f(f(b))). \end{array}$$

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

Example 9. Consider the following functional program reversing a list by means of an accumulating parameter:

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

Suppose we wish to transform $rev(xs)$. After two driving steps we have:



Here the rightmost leaf expression has the parent's expression embedded, so we should not drive the leaf. In fact, this decision is wise: repeated driving would never lead to an expression that is a renaming of an ancestor's expression.

The rationale behind using the homeomorphic embedding relation in this way is that in any infinite sequence $e_0, e_1, \ldots$ of expressions, there *definitely* are $i<j$ with $e_i \trianglelefteq e_j$ (see Theorem 51). 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.

The homeomorphic embedding relation is defined on elements of $\mathscr{E}$, not on elements $\mathscr{L}$. Therefore, in order to compare nodes in arbitrary trees over $\mathscr{L}$ we have to either extend the relation to $\mathscr{L}$ or make sure that it is not applied to elements of $\mathscr{L}\backslash\mathscr{E}$.

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. In fact, not only proper let-expressions, but all *trivial* expressions, will be handled this way.

Definition 12

1. An element of $\mathscr{L}$ is *trivial* if it has one of the following forms:
   (a) **let** $x_1 = e_1, \ldots, x_m = e_m$ **in** e where $m>0$;
   (b) $c(e_1, \ldots, e_n)$, where $n>0$;
2. Given $t \in T(\mathscr{L})$, a $\beta \in \mathrm{dom}(t)$ is *trivial* if $t(\beta)$ is trivial. Also,

$$\mathrm{triv}(t) = \{\beta \in \mathrm{dom}(t) \mid \beta \text{ is trivial}\}.$$

New leaf expressions, resulting from driving a node with a trivial expression, are strictly smaller than the former expression in a certain order (see Lemma 56).

Informally, this explains why there is no harm done in driving leaves with trivial expressions without comparing to ancestors.

The idea that a leaf node be compared to only *some* of its ancestors 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 trivial leaf is the empty set, and the relevant ancestors of a non-trivial leaf is all its non-trivial ancestors.

Definition 13. Let $t \in T(\mathscr{L})$ and $\beta \in \mathrm{dom}(t)$. The set $\mathrm{relanc}(t, \beta)$ of *relevant ancestors* of β in t is defined by

$$\mathrm{relanc}(t,\beta) = \begin{cases} \{\} & \text{if } \beta \in \mathrm{triv}(t) \\ \mathrm{anc}(t,\beta) \backslash \mathrm{triv}(t) & \text{if } \beta \notin \mathrm{triv}(t) \end{cases}$$

In conclusion, given a tree $t \in T(\mathscr{L})$ we may drive a $\beta \in \mathrm{leaf}(t)$ provided no relevant ancestor has an expression which is homeomorphically embedded in the leaf's expression. In the next subsection we present the generalization operations to be performed when some relevant ancestor *does* have an expression which is homeomorphically embedded in the leaf's expression.

4.4. Generalization: how

In generalization steps one compares two expressions and extracts common structure. For instance, in Example 2 we compared the root expression $a(a(xs, ys), xs)$ with the leaf expression $a(a(us, ys), u : us)$ and extracted the common structure $a(a(us, ys), zs)$.

The *most specific generalization* (see [15]) extracts the most structure in a certain sense.

Definition 14. Let $e_1, e_2 \in \mathscr{E}_H(V)$, for some H, V.

1. The expression e_2 is an *instance* of e_1, $e_1 \leqslant e_2$, if $e_1\theta = e_2$ for a substitution θ.
2. The expression e_1 is a *renaming* of e_2, $e_1 \doteq e_2$, if $e_1 \leqslant e_2$ and $e_1 \leqslant e_1$.
3. A *generalization* of e_1, e_2 is a expression e_g such that $e_g \leqslant e_1$ and $e_g \leqslant e_2$.
4. 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 $e'_g \leqslant e_g$.

Example 10. Let $x, y, u, v \in V$ and $f \in H$, and consider elements of $\mathscr{E}_H(V)$. Examples of renamings:

1. $f(x, y)$ is a renaming of $f(x, y)$;
2. $f(u, v)$ is a renaming of $f(x, y)$;
3. $f(y, x)$ is a renaming of $f(x, y)$.

Non-examples of renamings:

1. $f(f(u,v),x)$ is not a renaming of $f(x,y)$;
2. $f(x,x)$ is not a renaming of $f(x,y)$;
3. $f(x,y)$ is not a renaming of $f(x,x)$.

Remark. Note that we now use the term *generalization* in two distinct senses: to denote certain operations on trees performed by supercompilation (as in 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 11. Let $x, y \in V$, and $b, c, f \in H$. The following table gives examples of most specific generalizations e_g of $e_1, e_2 \in \mathscr{E}_H(V)$ and accompanying substitutions θ_1, θ_2 with $e_g\theta_i = e_i$:

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

Remark. Any two $e_1, e_2 \in \mathscr{E}_H(V)$ have at most one msg up to renaming; that is, if e_g and e'_g are both msg's of e_1 and e_2, then $e_g \doteq e'_g$.

Proposition 15. *For any $e_1, e_2 \in \mathscr{E}_H(V)$ there is an $e_g \in \mathscr{E}_H(V)$ and substitutions θ_1, θ_2 such that*:

1. *e_g is an msg of e_1 and e_2*;
2. *$e_g\theta_1 = e_1$ and $e_g\theta_2 = e_2$*;
3. $\mathrm{support}(\theta_1) = \mathrm{support}(\theta_2) = \mathrm{vars}(e_g)$.

Proof. See, e.g. [15]. □

Definition 16. Let $e_1, e_2 \in \mathscr{E}_H(V)$.

1. By $e_1 \sqcap e_2$ we denote a triple $(e_g, \theta_1, \theta_2)$ satisfying the conditions of Proposition 15.
2. We say that e_1 and e_2 are *incommensurable*, $e_1 \leftrightarrow e_2$, if $e_1 \sqcap e_2 = (x, \theta_1, \theta_2),\ x \in V$.

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

The generalization step in Example 2 is an example of an upwards 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 expressison which captures the common structure of the leaf and ancestor expressions. This common structure is computed by the most specific generalization operation.

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

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

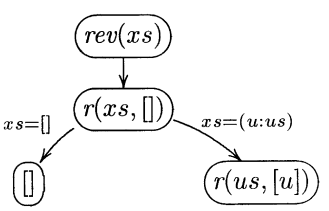
In such cases a split step is performed. The idea behind a split step is that if the ancestor expression is embedded in the leaf expression, then there is a subterm of the leaf expression which has structure in common with the ancestor. Hence, the split step digs out this structure.

The following example illustrates upwards and downwards abstract steps, and the next example illustrates split steps.

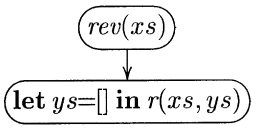
Example 12. Consider again the following functional programs reversing a list by means of an accumulating parameter:

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

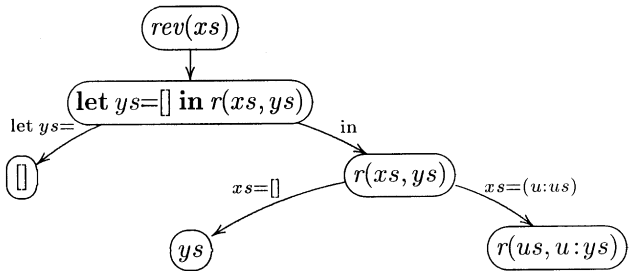
Again we transform $rev(xs)$, and after two driving steps we have



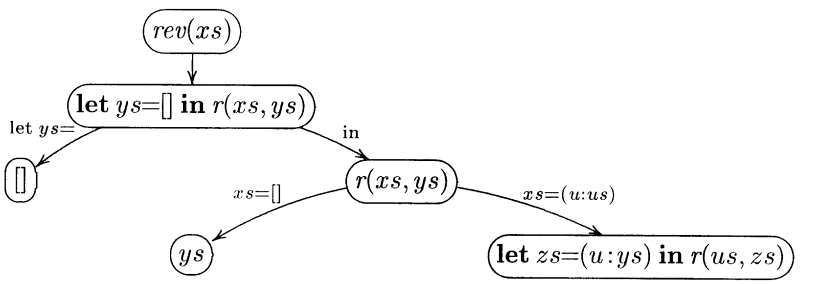
Here the rightmost leaf expression has its parent's expression embedded. We perform an upwards generalization step:



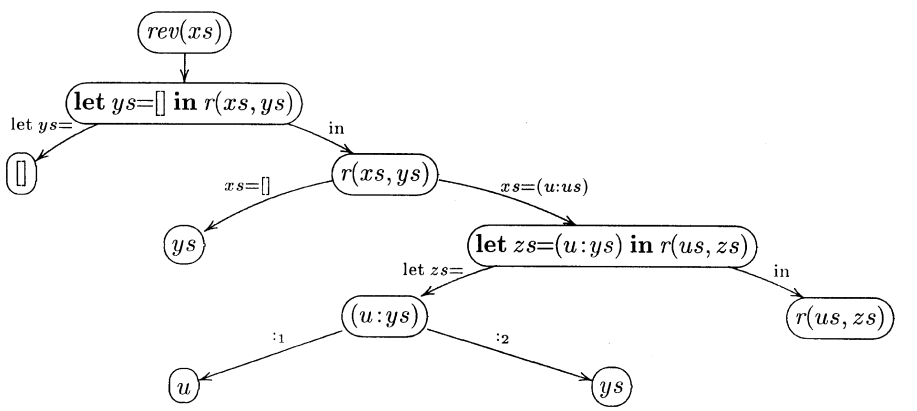
A few more driving steps yield:



Here the rightmost leaf expression has its parent's expression embedded. However, the leaf expression is an instance of its parent's expression. We therefore perform a downwards generalization step:



Driving finally leads to the following tree:

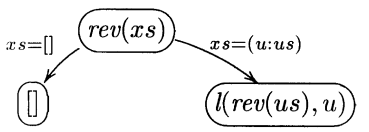


From this tree one can construct a new term and program (which turn out to be identical to the original term and program).

Example 13. Consider another functional program reversing a list:

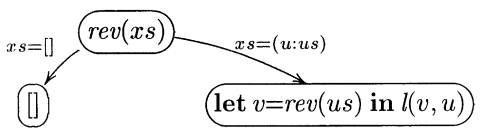
$$\begin{aligned} rev([]) &= [] \\ rev(u:us) &= l(rev(us),u) \\ l([],v) &= [v] \\ l(u:us,v) &= u:l(us,v) \end{aligned}$$

Suppose we wish to transform the expression $rev(xs)$. After the first driving step we have

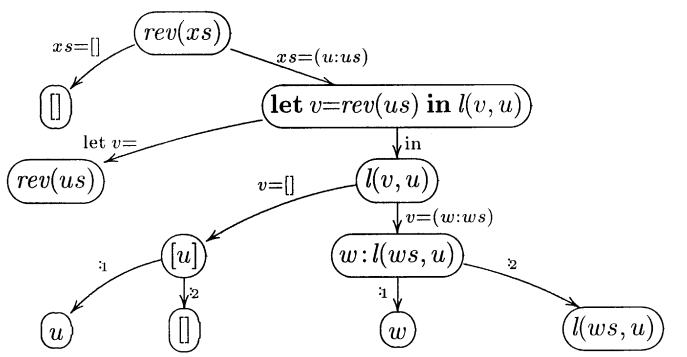


The root expression is embedded in the rightmost leaf expression. We cannot perform an upwards abstract step, since the msg of the two expressions is a variable. For the same reason we cannot perform a downwards abstract step.

However, the root expression clearly has structure in common with a subexpression of the leaf expression, namely the subexpression $rev(us)$. Hence we perform a split step:



Driving then gives



from which a new term and program can be recovered (that turn out to be identical to the original term and program).

The following, then, are the generalization operations used in positive supercompilation; the operations are illustrated in Fig. 2.

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

1. For $\beta \in \text{leaf}(t)$ with $t(\beta) = h(e_1, \ldots, e_n)$, $h \in C \cup F \cup G$, 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 $\alpha, \beta \in \text{dom}(t)$ with $t(\alpha), t(\beta) \in \mathscr{E}$, $t(\alpha) \sqcap t(\beta) = (e, \{x_1 := e_1, \ldots, x_n := e_n\}, \theta_2)$ define

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

Remark. Note that the abstract operation is defined only in case $t(\alpha), t(\beta) \in \mathscr{E}$ (not in general for $t(\alpha), t(\beta) \in \mathscr{L}$). This is fortunate since $\sqcap$ is defined only on $\mathscr{E}$. But will we not need to invoke the operations in cases where $t(\alpha), t(\beta) \in \mathscr{L} \backslash \mathscr{E}$? No: all $l \in \mathscr{L} \backslash \mathscr{E}$ are trivial and will hence be driven without comparison with ancestors.

4.5. Positive supercompilation

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

Definition 18. Let $t \in T_\infty(\mathscr{L})$. A $\beta \in \text{leaf}(t)$ is *processed* if β is non-trivial and 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. $t(\beta)$ is a renaming of $t(\alpha)$ for some $\alpha \in \text{relanc}(t, \beta)$.

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

Positive supercompilation $P : T(\mathscr{L}) \rightarrow T(\mathscr{L})$ can then be defined as follows.[6]

Definition 19. Given $t \in T(\mathscr{L})$, if t is closed $P(t) = t$. Otherwise, let $\beta \in \text{leaf}(t)$ be an unprocessed node and proceed as follows.

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

[6] 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.

Example 14. The steps in Examples 12 and 13 are exactly the steps that P computes.

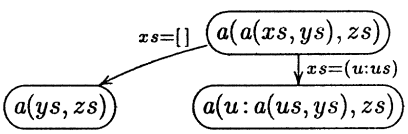
Remark. The algorithm calls abstract and split only in cases where these operations are well-defined. Indeed, when abstract (t, β, α) is called, then $\alpha \in \text{relanc}(t, \beta)$. In particular, α, β are non-trivial, so $t(\alpha)$, $t(\beta) \in \mathscr{E}$. Similarly, when abstract (t, α, β) is called. Finally, when split (t, β) is called, then β is non-trivial and unprocessed, so $t(\beta) \in \mathscr{E} \backslash X$, i.e., $t(\beta) = h(e_1, \ldots, e_n)$ for some $h \in C \cup F \cup G$.

In Section 7 we prove that P terminates.

In the next two subsections we introduce two variations of the above positive supercompiler. In the first, the relation $\trianglelefteq$ is replaced by another relation; in the other, the definition of relanc is changed.

4.6. *Positive supercompilation with very simple characteristic trees*

In some cases the above algorithm generalizes where one would have preferred it to drive. For instance, on the tree:



we cannot drive at the rightmost node since the root expression is embedded in its expression. In fact, the algorithm performs an upwards abstract step which separates the inner and outer call to append, in effect preventing elimination of the intermediate data structure.

The reason that we want to drive the leaf node, despite the fact that its expression has an ancestor's expression embedded, is that the ancestor's expression gave rise to several children corresponding to different patterns, whereas the leaf expression does not give rise to several children according to different patterns. In other words, new information is available in the leaf expression, and it is desirable that this be taken into account by a driving 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 [26] and others.

Definition 20. Define $B : \mathscr{E} \to \mathbb{B}$ by

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

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

Example 15.
1. $B(r(xs,[])) = B(r(us,[u]))$, so $r(xs,[]) \trianglelefteq^* r(us,[u])$;
2. $B(a(a(xs,ys),zs)) > B(a(u:a(us,ys),zs))$, so $a(a(xs,ys),zs) \not\trianglelefteq^* a(u:a(us,ys),zs)$.

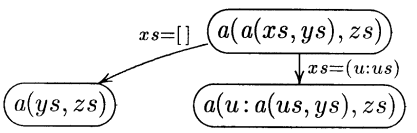
The above algorithm can then be repeated with $\trianglelefteq^*$ in place of $\trianglelefteq$. This algorithm will be called C.

Example 16. The algorithm computes exactly the sequences of trees in Examples 1–2.

In Section 7 we prove that C terminates.

4.7. Positive supercompilation with local unfolding

An alternative to our very simple characterisitic trees is to consider a form of *local unfolding* as adopted in partial deduction – see e.g. [26]. Recall again the tree from Example 1:



We wish to perform a driving step at the rightmost leaf without testing whether any ancestor's expression is embedded in the leaf expression (because, in fact, there is an ancestor whose expression is embedded in the leaf's expression: the root).

In the third variant of positive supercompilation, we will divide the non-trivial nodes into two categories: global ones and local ones. The *global* nodes are those that give rise to instantiation of variables in driving steps. For instance, in the above tree, the nodes labeled $a(a(xs,ys),zs)$ and $a(ys,zs)$ are global, whereas the one labeled $a(u : a(us,ys),zs)$ is not. The *local* nodes are the non-global ones.

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. Thus, in the above tree, we would not compare the rightmost leaf to the root.

Definition 21. Let $t \in T(\mathscr{L})$ and $\beta \in \mathrm{dom}(t)$ be non-trival.

1. Node β is *global* if $t(\beta) \rightarrow_\theta e$ for some $\theta \neq \{\}$. The set of *global ancestors* of β in t, $\mathrm{globanc}(t,\beta)$, is the set of global nodes in $\mathrm{anc}(t,\beta)$.
2. Node β is *local* if β is not global. The set of *immediate local ancestors* of β in t, $\mathrm{locanc}(t,\beta)$, is the set of local nodes among $\alpha_1,\ldots,\alpha_n$ $(n \geqslant 0)$, where $\alpha_1,\ldots,\alpha_n,\beta$ is the longest branch in t ending in β such that $t(\alpha_1),\ldots,t(\alpha_n)$ are all local or trivial.[7]

[7] Note that we compare a local leaf with local ancestors across trivial ancestors. This turns out to be necessary to ensure termination.

Definition 22. Let $t \in T(\mathscr{L})$ and $\beta \in \mathrm{dom}(t)$. The set of *relevant ancestors* of β in t, $\mathrm{relanc}(t,\beta)$, is defined by

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

The algorithm for P can then be repeated with the new definition of relevant ancestors. This algorithm will be called L.

Example 17. The algorithm computes exactly the trees in Examples 1 and 2.

Section 7 addresses termination of L.

5. The metric space of trees

As mentioned above, we will prove termination of our positive supercompilers in Section 7. However, first we shall develop in the next section a framework within which these proofs can be developed. The present section introduces a mathematical structure that is useful for this framework.

As suggested by the examples in Section 2, termination of an abstract program transformer amounts to a certain form of convergence of sequences of trees. We therefore first 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. [35]. Metric spaces have many applications in computer science – see e.g. [28, 37].

Having introduced metric spaces, we then recall how the set of trees over some set can be viewed as a metric space. This enables us to reason about convergence of our sequences of trees. Early papers studying the metric space of trees include [2, 3, 6–9, 12, 30]. More recent references appear in [28, 37]. Lloyd [27] uses the metric space of trees to present complete Herbrand interpretations for non-terminating logic programs.

5.1. Metric spaces

We first recall the concept of a *metric space*.

Definition 23. Let X be a set and $d : X \times X \to \mathbb{R}_+$ a map[8] with, for all $x, y, z \in X$:

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

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

[8] $\mathbb{R}_+ = \{r \in \mathbb{R} \mid r \geqslant 0\}$.

Example 18.

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

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

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

The following definition recalls the notion of a *convergent* sequence in a metric space. Informally, a sequence converges to some limit, if the distance between the limit and the elements of the sequence approach 0. A trivial special case, which will be useful in our framework, is when the elements of the sequence are identical to the limit from some step. In this case we say that the sequence *stabilizes* to the limit.

A slightly weaker property of sequences than being convergent is being a *Cauchy* sequence. Very informally, a sequence is Cauchy, if the distance between its elements approach 0. A metric space in which all Cauchy sequences are convergent is called *complete*. It is well-known that the metric space of trees is complete – see below – and this result turns out to be useful for our framework.

Definition 24. Let (X, d) be a metric space.

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

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

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

In our framework we shall consider predicates on the elements (trees) of certain sequences, and these predicates must satisfy certain well-behavedness conditions. The following well-known concepts will be useful for expressing these conditions.

Definition 26. Let $(X,d),(Y,d')$ be metric spaces. A map $f:X\to Y$ is *continuous at*[9] $x\in X$ if, for every sequence $x_0,x_1,\ldots\in X$ that converges to x, $f(x_0),f(x_1),\ldots\in Y$ converges to $f(x)$. Also, $f:X\to Y$ is *continuous* if f is continuous at every $x\in X$.

Example 19. 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 $p:X\to\mathbb{B}$, 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 $x_0,x_1,\ldots$ that converges to x, the sequence $p(x_0),p(x_1),\ldots$ converges to $p(x)$.

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

5.2. The metric space of trees

We now show that the set $T_\infty(E)$, for some set E, can be viewed as a metric space.

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 27. Define $d:T_\infty(E)\times T_\infty(E)\to\mathbb{R}_+$ 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).

The following shows how the notions of stabilizing, convergent, and Cauchy sequence appear in the special metric space $(T_\infty(E),d)$.

Remark

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

The next result, which was mentioned above, was first proved by Bloom et al. [7], and independently noted by Mycielski and Taylor [30] and Arnold and Nivat [2, 3].

Proposition 28. *The metric space $(T_\infty(E),d)$ is complete.*

[9] This is not the usual definition of continuity, but it is well-known that this definition is equivalent to the usual one.

Proof. Let $t_0, t_1, \ldots \in T_\infty(E)$ be a Cauchy sequence. Let $\mu(\ell)$ be the smallest number N such that for all $n \geqslant N$, all the initial subtrees $t_n[\ell], t_{n+1}[\ell], \ldots$ are identical; by the preceding remark, μ is well-defined. In symbols:

$$\mu(\ell) = \min\{N \in \mathbb{N} \mid \forall n \geqslant N\colon\ t_n[\ell] = t_{n+1}[\ell]\}$$

Then define a limit t as follows. For every ℓ, the initial subtrees $t_0[\ell], t_1[\ell], \ldots$ are identical from some step. The root of t is the root of the identical initial subtrees of depth 0. The nodes of depth 1 in t, if any, are the nodes at depth 1 of the identical initial subtrees of depth 1, etc. In symbols:

$$\begin{aligned} \mathrm{dom}(t) &= \{\alpha \in \mathbb{N}_1^* \mid \alpha \in \mathrm{dom}(t_{\mu(|\alpha|)})\} \\ t(\alpha) &= t_{\mu(|\alpha|)}(\alpha) \quad \text{for all } \alpha \in \mathrm{dom}(t) \end{aligned}$$

Then $t_0, t_1, \ldots$ converges to t. □

The following connection between stability, convergence, and predicates does not hold in arbitrary metric spaces. The result will be useful in the next section.

Lemma 29. A predicate p on $T_\infty(E)$ is continuous iff for every convergent sequence $t_0, t_1, \ldots \in T_\infty(E)$ with *infinite* limit t, the sequence $p(t_0), p(t_1), \ldots$ stabilizes to $p(t)$.

Proof. The left to right direction is obvious. For the other direction, assume the sequence $t_0, t_1, \ldots \in T_\infty(E)$ converges to t. We must prove that $p(t_0), p(t_1), \ldots$ converges to $p(t)$. If t is infinite, this follows from the assumptions. If t is finite, then $t_0, t_1, \ldots$ in fact stabilizes to t, so $p(t_0), p(t_1), \ldots$ stabilizes to $p(t)$. □

6. Termination of transformers

We now develop or framework for proving termination of abstract program transformers. In the first subsection, we give a condition ensuring termination of an abstract program transformer. In the next two subsections we consider some specific techniques for satisfying the condition.

6.1. A condition ensuring termination of apts

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

Our condition requires that the predicate maintained by the transformer be false on infinite trees.

Definition 31. A predicate $p : T_\infty(E) \to \mathbb{B}$ is *finitary* if $p(t) = 0$ for all infinite $t \in T_\infty(E)$.

Definition 32. An apt M on E is *Cauchy* if, for every singleton $t \in T_\infty(E)$, the sequence $t, M(t), M^2(t), \ldots$ is a Cauchy sequence.

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

Theorem 33. *Let apt* $M : T(E) \to T(E)$ *maintain predicate* $p : T_\infty(E) \to \mathbb{B}$. *If*

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

then M *terminates.*

Proof. Let apt $M : T(E) \to T(E)$ and predicate $p : T_\infty(E) \to \mathbb{B}$ satisfy the conditions of the theorem. Given some singleton $t \in T(E)$, consider the sequence

$$t_0, t_1, \ldots$$

where $t_i = M^i(t)$, By assumption this sequence is Cauchy. By completeness the sequence then converges to some $t \in T_\infty(E)$.

Suppose $t_0, t_1, \ldots$ is not *bounded*, i.e. for all ℓ, I, there exists $i \geqslant I$ such that $|t_i| > \ell$. Then t must be infinite. Hence $p(t)$ is false. Then, by continuity and Lemma 29, $p(t_n)$ is false for all $n \geqslant N$ for some N. This contradicts the assumption that M maintains p. Thus, $t_0, t_1, \ldots$ is bounded, i.e. there exists certain ℓ, I, such that for all $i \geqslant I, |t_i| \leqslant \ell$.

Since $t_0, t_1, \ldots$ is Cauchy, there exists J such that, for all $j \geqslant J$, $t_j[\ell] = t_{j+1}[\ell]$. With $N = \max\{I, J\}$ it follows that, for all $n \geqslant N$,

$$\begin{aligned} t_n &= t_n[\ell] && \text{since } |t_n| \leqslant \ell \\ &= t_{n+1}[\ell] && \text{since } t_n[\ell] = t_{n+1}[\ell] \\ &= t_{n+1} && \text{since } |t_{n+1}| \leqslant \ell \end{aligned}$$

Thus, $t_0, t_1, \ldots$ stabilizes, so M terminates. □

The proof shows that the following slightly stronger result holds.

Definition 34. Let $M : T(E) \to T(E)$ be an apt on E and $p : T_\infty(E) \to \mathbb{B}$ be a predicate. M *weakly maintains* p if, for every singleton $t \in T(E)$ it holds that $p(M^i(t)) = 1$ for infinitely many $i \in \mathbb{N}$.

Corollary 35. *Let apt* $M : T(E) \to T(E)$ *weakly maintain* $p : T_\infty(E) \to \mathbb{B}$. *If*

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

then M terminates.

In other words, the transformer may make some intermediate steps in which its predicate is temporarily false, as long as it always eventually returns to a state where the predicate is true again.

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.

In the rest of this section we consider specific techniques for ensuring that an apt is Cauchy and that a predicate is finitary and continuous.

6.2. Cauchy transformers

We begin by studying circumstances under which a transformer is Cauchy. The following two definitions fix terminology for some well-known concepts.

Definition 36. Let S be a set with a relation $\leqslant$. Then $(S, \leqslant)$ is a *quasi-order* if $\leqslant$ is reflexive and transitive. We write $s < s'$ if $s \leqslant s'$ and $s' \not\leqslant s$.

Definition 37. Let $(S, \leqslant)$ be a quasi-order.

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

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

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

Then M is Cauchy.

Proof. Given a singleton tree t, let $t_i = M^i(t)$. We prove by induction on ℓ that, for all ℓ, there is N such that, for all $m, n \geqslant N$, $t_n[\ell] = t_m[\ell]$.

For $\ell = 0$, suppose $\{t_i(\varepsilon) \mid i \in \mathbb{N}\}$ is infinite. Then there are infinitely many generalization steps, i.e. for infinitely many i, $t_i(\varepsilon) > t_{i+1}(\varepsilon)$. This clearly contradicts the assumption that $\leqslant$ is a well-founded quasi-order. Hence there is an N_0 such that for all $n, m \geqslant N_0$, $t_n(\varepsilon) = t_m(\varepsilon)$, i.e. $t_n[0] = t_m[0]$.

For $\ell > 0$ there is, by the induction hypothesis, an $N_{\ell-1}$ such that for all $m, n \geqslant N_{\ell-1}$, $t_n[\ell - 1] = t_m[\ell - 1]$. The only way children can be added to level ℓ after step $N_{\ell-1}$ is by an unfolding step. Thus, there is a number M such that the number of children at level ℓ is the same in t_n for all $n \geqslant M$. Let this number of children be K. For each node at level ℓ now proceed as in the case $\ell = 0$. This gives K numbers $N_1, \ldots, N_K \geqslant M$. Let $N = \max\{N_1, \ldots, N_K\}$. Then, for all m, $n \geqslant N$, $t_n[\ell] = t_m[\ell]$. □

6.3. Continuous predicates

Now 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. Adapting the notion to families of *trees*, we might say that a family $T \subseteq T_\infty(E)$ of trees is of finite character if for all $t \in T_\infty(E)$ it holds that $t \in T$ if and only if for all $\ell \in \mathbb{N}$: $t[\ell] \in T$. Identifying a predicate $p : T_\infty(E) \to \mathbb{B}$ with the family $\{t \in T_\infty(E) \mid p(t) = 1\}$ we arrive at the following definition.

Definition 39. A predicate $p : T_\infty(E) \to \mathbb{B}$ is of *finite character* iff, for all $t \in T_\infty(E)$:

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

Remark. Perhaps the equivalence

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

is easier to recall in the form:

$$p(t) = 1 \Rightarrow \forall \ell \in \mathbb{N} : p(t[\ell]) = 1$$
$$p(t) = 0 \Rightarrow \exists \ell \in \mathbb{N} : p(t[\ell]) = 0.$$

Proposition 40. *Suppose $p : T_\infty(E) \to \mathbb{B}$ is finitary and is of finite character. Then p is continuous.*

Proof. Let $t_0, t_1, \ldots$ converge to an infinite limit t. By Lemma 29 it suffices to show that $p(t_0), p(t_1), \ldots$ stabilizes to $p(t)$.

Since p is finitary, $p(t) = 0$. By assumption there is an ℓ such that already $p(t[\ell]) = 0$. Since $t_0, t_1, \ldots$ converges to t, there exists an N such that, for all $n \geqslant N$, $t_n[\ell] = t[\ell]$. Therefore, for all $n \geqslant N$, $p(t_n[\ell]) = 0$. By assumption, then also $p(t_n) = 0$. □

We end the section by reviewing instances of Proposition 40.

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, 19, 26, 38, 45] – see also Section 7.

Proposition 41. *Let* $(E, \leqslant)$ *be a well-quasi-order. Then* $p : T_\infty(E) \to \mathbb{B}$,

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

is finitary and continuous.

Proof. We first prove that p is finitary. An infinite tree t has, by König's Lemma (remember that trees are finitely branching), an infinite branch, and since $\leqslant$ is a well-quasi-order, there must be a node α and a subsequent node $\alpha i\beta$ with $t(\alpha) \leqslant t(\alpha i\beta)$, so $p(t)$ is false.

To prove continuity, we use Proposition 40. If $p(t) = 1$, then clearly also $p(t[\ell]) = 1$ for all ℓ. Moreover, if for all ℓ it holds that $p(t[\ell]) = 1$, then also $p(t) = 1$; indeed, if $p(t) = 0$, i.e. $t(\alpha) \leqslant t(\alpha i\beta)$ for some $\alpha, \alpha i\beta \in \mathrm{dom}(t)$, then already $p(t[\ell]) = 0$ where $\ell = |\alpha i\beta|$. □

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 42. *Let* $(E, \leqslant)$ *be a well-founded quasi-order. Then* $p : T_\infty(E) \to \mathbb{B}$,

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

is finitary and continuous.

Proof. Similar to the proof of Proposition 41. □

Remark. Another formulation of the predicate in the preceding proposition is the following:

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

Indeed, when $p(t) = 0$ then also $p'(t) = 0$ (take $\beta = \varepsilon$). Conversely, if $p'(t) = 0$ then $t(\alpha) \not> t(\alpha i\beta)$ for some α, i, β. Consider the nodes $\alpha, \alpha i, \ldots, \alpha i\beta$ on the branch from α to $\alpha i\beta$. We cannot have

$$t(\alpha) > t(\alpha i) > \cdots > t(\alpha i\beta)$$

because this would entail $t(\alpha) > t(\alpha i\beta)$. Thus there must be a node γ and a child γj with $t(\gamma) \not> t(\gamma j)$. Therefore $p(t) = 0$.

In the following definition, a formulation similar to p' is used.

The following generalization of the preceding proposition is used in some techniques for ensuring global termination of partial deduction [29].

Proposition 43. *Let* $\{E_1,\ldots,E_n\}$ *be a partition*[10] *of* E *and* $\leqslant_1,\ldots,\leqslant_n$ *be well-founded quasi-orders on* $E_1,\ldots,E_n$, *respectively. Then* $p:T_\infty(E)\to\mathbb{B}$,

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

is finitary and continuous.

Proof. Similar to the proof of Proposition 41. □

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

Proposition 44. *Let* $\{E_1,E_2\}$ *be a partition of* E *and let* $\leqslant_1$ *be a well-quasi-order on* E_1 *and* $\leqslant_2$ *a well-founded quasi-order on* E_2. *Then* $p:T_\infty(E)\to\mathbb{B}$,

$$p(t)=\begin{cases}0 & \text{if } \exists\alpha,\alpha i\beta\in\mathrm{dom}(t):t(\alpha),t(\alpha i\beta)\in E_1\ \&\ t(\alpha)\leqslant_1 t(\alpha i\beta),\\ 0 & \text{if } \exists\alpha,\alpha i\in\mathrm{dom}(t):t(\alpha),t(\alpha i)\in E_2\wedge t(\alpha)\not\geqslant_2 t(\alpha i),\\ 1 & \text{otherwise},\end{cases}$$

is finitary and continuous.

Proof. Similar to the proof of Proposition 41. □

The following shows that it suffices to apply a finitary and continuous predicate to the *interior* part of a tree; that is, the leaves are not important.

Definition 45. For $t\in T_\infty(E)$, defines the *interior* $t^0\in T_\infty(E)$ of t by

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

Proposition 46. *Let* $p:T_\infty(E)\to\mathbb{B}$ *be finitary and continuous. Then also the map* $q:T_\infty(E)\to\mathbb{B}$ *defined by*

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

is finitary and continuous.

Proof. Let $p:T_\infty(E)\to\mathbb{B}$ be finitary and continuous and define q by $q(t)=p(t^0)$.

[10] That is, $E_1,\ldots,E_n$ are sets with $\bigcup_{i=1}^n E_i=E$ and $i\neq j\Rightarrow E_i\cap E_j=\emptyset$.

For infinite $t \in T_\infty(E)$, $t^0 \in T_\infty(E)$ is also infinite, so $q(t) = p(t^0) = 0$, and hence q is finitary.

To show that q is continuous it suffices to show that the interior projection $\bullet^0 : T_\infty(E) \to T_\infty(E)$ is continuous. For this end, first note, that for all t, $u \in T_\infty(E)$ and all $\ell \in \mathbb{N}$:

$$t[\ell+1] = u[\ell+1] \Rightarrow t^0[\ell] = u^0[\ell]$$

Now, if $t_1, t_2, \ldots$ converges to t, there is, for any $\ell \in \mathbb{N}$, and $N \in \mathbb{N}$ such that for all $n \geqslant N$, $t_n[\ell+1] = t[\ell+1]$, i.e. $t_n^0[\ell] = t^0[\ell]$, so $t_1^0, t_2^0, \ldots$ converges to t^0, as required. □

Remark. The map $\bullet^0$ is continuous for a special reason: it satisfies a *Lipschitz condition.*[11]

It is not hard to see that one can replace $\bullet^0$ in Proposition 46 by any continuous map which maps infinite trees to infinite trees.

The following result states that it suffices to apply a finitary and continuous predicate to the immediate subtrees of a tree; that is, that the root is not important.

Definition 47. For $t \in T_\infty(E)$, the set of *immediate subtrees of t* is the possibly empty set $\{s_1, \ldots, s_n\}$ such that

$$\mathrm{dom}(t) = \{\varepsilon\} \cup \{1\alpha \mid \alpha \in \mathrm{dom}(s_1)\} \cup \cdots \cup \{n\alpha \mid \alpha \in \mathrm{dom}(s_n)\},$$
$$t(i\alpha) = s_i(\alpha).$$

We also say that t has *arity* n and write $\pi_i(t)$ for s_i for each $i \in \{1, \ldots, n\}$.

Proposition 48. *Let $p : T_\infty(E) \to \mathbb{B}$ be finitary and continuous. Then also the map $q : T_\infty(E) \to \mathbb{B}$ defined by*[12]

$$q(t) = p(s_1) \wedge \cdots \wedge p(s_n),$$

where $\{s_1, \ldots, s_n\}$ are the immediate subtrees of t, is finitary and continuous.

Proof. Let $p, q : T_\infty(E) \to \mathbb{B}$ be defined as stated above.

To see that q is finitary, consider some infinite $t \in T_\infty(E)$. Then t has immediate subtrees $\{s_1, \ldots, s_n\}$ where at least one s_i is infinite. Then $p(s_i) = 0$, so also $q(t) = 0$, as required.

To prove continuity, let $t_0, t_1, \ldots$ converge to infinite limit t. For some $N \in \mathbb{N}$ the trees $t_N, t_{N+1}, \ldots$ all have the same arity n, which is also the arity of t. Moreover, for each $i \in \{1, \ldots, n\}$ the immediate subtrees $\pi_i(t_N), \pi_i(t_{N+1}), \ldots$ converge to $\pi_i(t)$.

[11] A map $f : X \to Y$ from one metric space (X, d_X) to another (Y, d_Y) satisfies a Lipschitz condition if there is an $r \in \mathbb{R}_+$ such that for any $x, x' \in X : d_Y(f(x), f(x')) \leqslant r \cdot d_X(x, x')$. Exercise: What does this amount to in the specific metric space of trees?

[12] We define $b_1 \wedge \cdots \wedge b_n = 0$ if there is an $i \in \{1, \ldots, n\}$ with $b_i = 0$, and $b_1 \wedge \cdots \wedge b_n = 1$ otherwise. In particular, $b_1 \wedge \cdots \wedge b_n = 1$, if $n = 0$.

For at least one $i, \pi_i(t)$ is infinite, so $p(\pi_i(t)) = 0$, hence $q(t) = 0$. By continuity, $p(\pi_i(t_N)), p(\pi_i(t_{N+1})), \ldots$ converges to $p(\pi_i(t))$, so for some K, we have that $p(\pi_i(t_K)) = 0$, $p(\pi_i(t_{K+1})) = 0, \ldots$. Thus, $q(t_K) = 0$, $q(t_{K+1}) = 0, \ldots$, i.e. $q(t_0), q(t_1), \ldots$ converges to $q(t)$, as required. □

7. Application: termination of positive supercompilation

In this section we address termination of the three variants of positive supercompilation. We first prove that positive supercompilation P terminates. We do so by proving that P is Cauchy and that P maintains a finitary, continuous predicate; the desired result then follows by Theorem 33. This occupies the first two subsections. The last section considers the variants C and L as well as an algorithm for partial deduction.

7.1. P is Cauchy

We now prove that P is Cauchy; the idea is to use Proposition 38. Indeed, P always either unfolds in a driving step or replaces in a generalization step a subtree by a new leaf. The root label of the former subtree is in $\mathscr{E}$, whereas the new leaf's label is in $\mathscr{L}\backslash\mathscr{E}$. Thus, if we count elements of $\mathscr{E}$ as larger than elements of $\mathscr{L}\backslash\mathscr{E}$, then we have a measure that strictly decreases in generalization steps. The main problem is then to show that the let-expressions introduced in generalization steps are proper, i.e. that they really belong to $\mathscr{L}\backslash\mathscr{E}$.

We will need the following lemma.

Lemma 49. *Let $e_g \in \mathscr{E}_H(V)$ be an msg of e_1 and e_2. Then*

$$e_1 \leqslant e_2 \;\Leftrightarrow\; e_1 \doteq e_g.$$

Proof. Routine verification. □

Proposition 50. *P is Cauchy.*

Proof. Define the relation $\succcurlyeq$ on $\mathscr{L}$ by

$$\ell \succcurlyeq \ell' \;\Leftrightarrow\; \ell \in \mathscr{E} \vee \ell' \notin \mathscr{E}.$$

We have $\ell \succ \ell'$ iff $\ell \in \mathscr{E}$ and $\ell' \notin \mathscr{E}$. In other words, replacing an improper let-expression (an element of $\mathscr{E}$) by a proper let-expression (an element of $\mathscr{L}\backslash\mathscr{E}$) strictly decreases the order. It is a routine exercise to verify that $\succcurlyeq$ is a well-founded quasi-order.

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

$$P(t) = t\{\gamma := t'\}$$

where, for some $\gamma \in \mathrm{dom}(t)$ and $t' \in T(\mathscr{L})$, either $\gamma \in \mathrm{leaf}(t)$ and $t(\gamma) = t'(\varepsilon)$, or $t(\gamma) \succ t'(\varepsilon)$. We proceed by case analysis of the operation performed by P.

1. $P(t) = \mathrm{drive}(t, \gamma) = t\{\gamma := t'\}$, where $\gamma \in \mathrm{leaf}(t)$. In this case, for certain expressions $e_1, \ldots, e_n$, $t' = t(\gamma) \to e_1, \ldots, e_n$. Then

$$t(\gamma) = t'(\varepsilon).$$

2. $P(t) = \mathrm{abstract}(t, \gamma, \alpha) = t\{\gamma := \mathbf{let}\, x_1 = e_1, \ldots, x_n = e_n \,\mathbf{in}\, e \to\}$, where $\alpha \in \mathrm{anc}(t, \gamma)$ and $(t(\alpha) \trianglelefteq t(\gamma)$ (downwards abstract). Since γ is not processed, $t(\alpha) \neq t(\gamma)$. By definition of the abstract operation, $t(\gamma) = e\{x_1 := e_1, \ldots, x_n := e_n\}$. Since $t(\alpha) \trianglelefteq t(\gamma)$, also $t(\alpha) \doteq e$ by Lemma 49. Therefore $e \neq t(\gamma)$, so $n > 0$. Thus

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

3. $P(t) = \mathrm{abstract}(t, \gamma, \beta) = t\{\gamma := \mathbf{let}\, x_1 = e_1, \ldots, x_n = e_n \,\mathbf{in}\, e \to\}$, where $\gamma \in \mathrm{anc}(t, \beta)$ and $t(\gamma) \not\trianglelefteq t(\beta)$ (upwards abstract). By definition of the abstract operation, $t(\gamma) = e\{x_1 := e_1, \ldots, x_n := e_n\}$. Since $t(\gamma) \not\trianglelefteq t(\beta)$, also $t(\gamma) \neq e$ by Lemma 49. But $t(\gamma) = e\{x_1 := e_1, \ldots, x_n := e_n\}$, so $n > 0$. Thus

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

4. $P(t) = \mathrm{split}(t, \gamma) = t\{\gamma := \mathbf{let}\, x_1 = e_1, \ldots, x_n = e_n \,\mathbf{in}\, h(x_1, \ldots, x_n) \to\}$ where, for some $\alpha \in \mathrm{anc}(t, \gamma)$, $t(\alpha) \leftrightarrow t(\gamma)$. Since γ is non-trivial and unprocessed, $t(\gamma)$ must have form $h(e_1, \ldots e_n)$, where $h \in F \cup G$. Here $n > 0$: if $n = 0$, then $t(\gamma) = h()$. Since $t(\alpha) \trianglelefteq t(\gamma)$, also $t(\alpha) = h()$, contradicting $t(\alpha) \not\leftrightarrow t(\gamma)$. Thus,

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

Now use Proposition 38. □

7.2. *P maintains a finitary, continuous predicate*

Now we prove that P maintains a finitary, continuous predicate. The main idea is to prove that P maintains a predicate of the form in Proposition 44, where the well-quasi-order is $\trianglelefteq$ on non-trivial nodes (except possibly the leaves) and where we have a certain well-founded quasi-order on the trivial nodes (except possibly in the root).

The following result, known as *Kruskal's Tree Theorem*, is due to Higman [22] and Kruskal [25]. Its classical proof is due to Nash-Williams [31].

Theorem 51. *$(\mathscr{E}_H(V), \trianglelefteq)$ is a well-quasi-order, provided H is finite.*

Proof. Collapse all variables to one 0-ary operator and use the proof in [14]. □

In order to define the well-founded quasi-order on trivial expressions we need the following notions.

Definition 52

1. Define the size $|\bullet| : \mathscr{E} \to \mathbb{N}$ by

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

2. Define $1 : \mathscr{L} \to \mathscr{E}$ by

$$1(\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e) = e\{x_1 := e_1, \ldots, x_n := e_n\}.$$

(Here $n \geqslant 0$.)

3. Define $\sqsupseteq$ on $\mathscr{L}$ by

$$\ell \sqsupseteq \ell' \;\Leftrightarrow\; |l(\ell)| > |l(\ell')| \vee (|l(\ell)| = |l(\ell')| \wedge l(\ell) \cdot \geqslant l(\ell')).$$

Remark. We have

$$\ell \sqsupset \ell' \;\Leftrightarrow\; |l(\ell)| > |l(\ell')| \vee (|l(\ell)| = |l(\ell')| \wedge l(\ell) \geqslant l(\ell')).$$

For example,

$$\textbf{let } x = e \textbf{ in } c(x, y, z) \sqsupset \textbf{let } x = e \textbf{ in } c(x, y, y)$$

But

$$\textbf{let } x = e \textbf{ in } c(x, y) \not\sqsupset \textbf{let } x = e \textbf{ in } c(x, z)$$

although

$$\textbf{let } x = e \textbf{ in } c(x, y) \sqsupseteq \textbf{let } x = e \textbf{ in } c(x, z)$$

Lemma 53. *The relation $\sqsubseteq$ is a well-founded quasi-order.*

Proof. Routine verification using the fact that $\leqslant\cdot$ is a well-founded quasi-order. □

We would like to show that for all trivial $\ell \in \mathscr{L}$ it holds that $\ell \Rightarrow \ell'$ implied $\ell \sqsupset \ell'$. Unfortunately, this does not hold. In the following examples of $\ell \Rightarrow \ell'$, we have $\ell \sqsupseteq \ell'$, but $\ell \not\sqsupset \ell'$.

1. $\textbf{let } x = e \textbf{ in } x \Rightarrow e$;
2. $\textbf{let } x = e \textbf{ in } y \Rightarrow y$;
3. $\textbf{let } x = y \textbf{ in } x \Rightarrow x$.

However, we can prove that P never introduces such let-expressions. To this end, we define the following set.

Definition 54. The set $\mathscr{L}_0$ of *restricted let-expressions* is the set of all

$$\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e$$

where $n = 0$ or

1. $e \notin X$; and
2. $x_1, \ldots, x_n \in \mathrm{vars}(e)$; and
3. $e\{x_1 := e_1, \ldots, x_n := e_n\} \neq e$.

In other words, $\ell \in \mathscr{L}_0$ if $\ell \in \mathscr{E}$ or ℓ satisfies the above three conditions.

We can now prove that P always introduces let-expressions in $\mathscr{L}_0$. However, our aim is to show that P as an apt on $T_\infty(\mathscr{L})$ terminates, i.e. that P terminates on any singleton tree $t \in T_\infty(\mathscr{L})$. In this initial singleton, the label may be an $\ell \in \mathscr{L} \setminus \mathscr{L}_0$. This explains the exception concerning the root in the following lemma.

Lemma 55. *Given singleton* $t_0 \in T_\infty(\mathscr{L})$ *and* $i \in \mathbb{N}$, *let* $t' = P^i(t_0)$. *Then, for all* $\alpha \in \mathrm{dom}(t') \setminus \{\varepsilon\}$: $t(\alpha) \in \mathscr{L}_0$.

Proof. (By induction on i)

In the case $i = 0$ there is nothing to prove, since $P^0(t_0) = t_0$ is a singleton.

Now assume $i = j + 1$ where $j \geqslant 0$. Let $t = P^j(t)$. We split into cases according to the operation performed by P in the last step $t' = P(t)$.

1. $P(t) = \mathrm{drive}(t, \gamma) = t\{\gamma := t''\}$, where $\gamma \in \mathrm{leaf}(t)$. In this case, for certain expressions $e_1, \ldots, e_n$, $t'' = t(\gamma) \to e_1, \ldots, e_n$. The only new children are labeled $e_1, \ldots, e_n \in \mathscr{E}$ so these are also in $\mathscr{L}_0$.
2. $P(t) = \mathrm{abstract}(t, \gamma, \alpha) = t\{\gamma := \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \to\}$, where $\alpha \in \mathrm{anc}(t, \gamma)$ and $t(\alpha) \leqslant t(\gamma)$ (downwards abstract). We have to show that

   $$\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \in \mathscr{L}_0.$$

   We show that each of the three conditions in the definition of $\mathscr{L}_0$ are satisfied.

   (a) Since $t(\alpha) \leqslant t(\gamma)$, $e \doteq t(\alpha)$. Since $t(\alpha) \notin X$ (if $t(\alpha) \in X$, then α would have been processed and have received no children), also $e \notin X$.

   (b) We have $x_1, \ldots, x_n \in \mathrm{vars}(e)$ by definition of the abstract operation.

   (c) If $e\{x_1 := e_1, \ldots, x_n := e_n\} \doteq e$, then

   $$\begin{aligned} t(\gamma) &= e\{x_1 := e_1, \ldots, x_n := e_n\} \\ &\doteq e \\ &\leqslant t(\alpha) \end{aligned}$$

   and since also $t(\alpha) \leqslant t(\gamma)$, in fact $t(\alpha) \doteq t(\gamma)$, contradicting the fact that γ is not processed.

3. $P(t) = \text{abstract}(t, \gamma, \beta) = t\{\gamma := \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \rightarrow\}$, where $\gamma \in \text{anc}(t, \beta)$ and $t(\gamma) \not\trianglelefteq t(\beta)$ (upwards abstract). We have to show that

   $$\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \in \mathscr{L}_0.$$

   (a) Since $t(\gamma) \not\leftrightarrow t(\beta)$, the msg e of $t(\gamma)$ and $t(\beta)$ is not a variable, i.e. $e \notin X$.
   (b) We have $x_1, \ldots, x_n \in \text{vars}(e)$ by definition of the abstract operation.
   (c) If $e\{x_1 := e_1, \ldots, x_n := e_n\} \doteq e$, then

   $$\begin{aligned} t(\gamma) &= e\{x_1 := e_1, \ldots, x_n := e_n\} \\ &\doteq e \\ &\trianglelefteq t(\beta) \end{aligned}$$

   However, if this had been the case, then we would have performed a downwards abstract step.
4. $P(t) = \text{split}(t, \gamma) = t\{\gamma := \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } h(x_1, \ldots, x_n) \rightarrow\}$ where, for some $\alpha \in \text{anc}(t, \gamma)$, $t(\alpha) \leftrightarrow t(\gamma)$. We have to show that

   $$\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } h(x_1, \ldots, x_n) \in \mathscr{L}_0.$$

   (a) $h(x_1, \ldots, x_n) \notin X$.
   (b) $x_1, \ldots, x_n \in \text{vars}(h(x_1, \ldots, x_n))$.
   (c) Since $t(\alpha) \trianglelefteq t(\gamma) = h(e_1, \ldots, e_n)$ and $t(\alpha) \leftrightarrow t(\gamma)$, we have $t(\alpha) \trianglelefteq e_i$ for some i. Since $t(\alpha) \notin X$, also $e_i \notin X$. Hence $h(x_1, \ldots, x_n) \not\doteq h(e_1, \ldots, e_n)$.
   This concludes the proof. □

Our well-founded quasi-order decreases when we reduce on restricted, trivial expressions.

Lemma 56. *For all trivial $\ell \in \mathscr{L}_0$:*

$$\ell \Rightarrow \ell' \text{ implies } \ell \sqsupset \ell'.$$

Proof. Let $\ell \in \mathscr{L}_0$ be some trivial expression.

1. $\ell = c(e_1, \ldots, e_n) \Rightarrow e_i$. Then

   $$\begin{aligned} |l(\ell)| &= |c(e_1, \ldots, e_n)| \\ &= 1 + |e_1| + \cdots + |e_n| \\ &> |e_i| \\ &= |l(e_i)| \end{aligned}$$

   so $\ell \sqsupset e_i$.
2. $\ell = \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \Rightarrow \ell'$. We consider two cases.

(a) $\ell' = e_i$ for some i. In this case, notice that $x_i \in \mathrm{vars}(e)$ and $e \neq x_i$. Therefore,

$$\begin{aligned} |l(\ell)| &= |e\{x_1 := e_1, \ldots, x_n := e_n\}| \\ &> |e_i| \\ &= |l(e_i)| \end{aligned}$$

so $\ell \sqsupset e_i$.

(b) $\ell' = e$. Then clearly $|l(\ell)| \geqslant |l(\ell')|$. The e_i could all be variables or 0-ary constructors in which case $|l(\ell)| = |l(e)|$. Fortunately, $e\{x_1 := e_1, \ldots, x_n := e_n\}$ is not a renaming of e, that is, $e\{x_1 := e_1, \ldots, x_n := e_n\} \succcurlyeq e$, so

$$\begin{aligned} l(\ell) &= e\{x_1 := e_1, \ldots, x_n := e_n\} \\ &\succcurlyeq e_i \\ &= l(e_i) \end{aligned}$$

so again $\ell \sqsupset e$. □

Proposition 57. *P maintains a finitary, continuous predicate.*

Proof. Consider the predicate $q : T_\infty(\mathscr{L}) \to \mathbb{B}$ defined by

$$q(t) = p(s_1) \wedge \cdots \wedge p(s_n)$$

where $\{s_1, \ldots, s_n\}$ are the immediate subtrees of t^0, and where $p : T_\infty(\mathscr{L}) \to \mathbb{B}$ is defined by:

$$p(t) = \begin{cases} 0 & \text{if } \exists \alpha, \alpha i \beta \in \mathrm{dom}(t) : t(\alpha), t(\alpha i \beta) \text{ are non-trivial} \wedge\, t(\alpha) \trianglelefteq t(\alpha i \beta), \\ 0 & \text{if } \exists \alpha, \alpha i \in \mathrm{dom}(t) : t(\alpha), t(\alpha i) \text{ are trivial} \wedge\, t(\alpha) \not\sqsupseteq t(\alpha i), \\ 1 & \text{otherwise.} \end{cases}$$

The sets of non-trivial and trivial expressions constitute a partition of $\mathscr{L}$. Also, $\trianglelefteq$ is a well-quasi-order on the set of non-trivial expressions (in fact, on all of $\mathscr{E}$) and $\sqsubseteq$ is a well-founded quasi-order on the set of trivial expressions (in fact, on all of $\mathscr{L}$). It follows by Proposition 44 that p is finitary and continuous. By Proposition 48,

$$t \mapsto p(\pi_1(t)) \wedge \cdots \wedge p(\pi_n(T))$$

where n is the arity of t is also finitary and continuous. Finally, by Proposition 46, q is finitary and continuous.

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

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

(i) $t(\beta)$ non-trivial $\wedge \beta \notin \mathrm{leaf}(t) \Rightarrow \forall \alpha \in \mathrm{relanc}(t, \beta) : t(\alpha) \not\trianglelefteq t(\beta)$;

(ii) $\beta = \alpha j i \wedge t(\alpha j)$ trivial $\Rightarrow t(\alpha j) \sqsupset t(\beta)$.

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

It is easy to see that $q(t) = 1$ if t is good. The converse does not hold. For instance, for goodness we require $t(\alpha j) \sqsupset t(\alpha ji)$ when αj is trivial, even though αji is non-trivial. For q to be true, we only require $t(\alpha j) \sqsupset t(\alpha ji)$ when both αj and αji are trivial.

In other words, we strengthen the induction hypothesis. This is done to make it easier to prove that $t(\alpha j) \sqsupset t(\alpha ji)$ after a step that changes αji from being non-trivial to being trivial.

In conclusion, it suffices to show for any singleton $t_0 \in T_\infty(\mathscr{L})$ that $P^i(t_0)$ 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 P on $P^{i-1}(t_0)$. Before considering these cases, note that by the definition of goodness, if $t \in T_\infty(\mathscr{L})$ is good, $\gamma \in \mathrm{dom}(t)$, and $t' \in T_\infty(\mathscr{L})$, then $t\{\gamma := t'\}$ is good too, provided $\gamma\delta$ is good in $t\{\gamma := t'\}$ for all $\delta \in \mathrm{dom}(t')$.

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

1. $P(t) = \mathrm{drive}(t, \gamma) = t\{\gamma := t'\}$, where $\gamma \in \mathrm{leaf}(t)$, $t' = t(\gamma) \rightarrow e_1, \ldots, e_n$, and $\{e_1, \ldots, e_n\} = \{e \mid t(\gamma) \Rightarrow e\}$.
   We must show that $\gamma, \gamma 1, \ldots, \gamma n$ are good in $P(t)$.
   To see that γ is good in $P(t)$, 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 γi is good in $P(t)$, note that condition (i) is vacuously satisfied. Moreover, when $\ell \Rightarrow e$ and ℓ is trivial, also $\ell \sqsupset e$ by Lemmas 55 and 56, so condition (ii) holds as well.
2. $P(t) = \mathrm{abstract}(t, \gamma, \alpha) = t\{\gamma := \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \rightarrow\}$, where $\alpha \in \mathrm{anc}(t, \gamma)$.
   We must show that γ is good in $P(t)$. Condition (i) holds vacuously, and (ii) follows from the induction hypothesis and $l(t(\gamma)) = l(\textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e)$.

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

Theorem 58. *P terminates.*

Proof. By Theorem 33 and Propositions 50 and 57. □

To show that C terminates we need the following.

Corollary 59. *The relation $\trianglelefteq^*$ is a well-quasi order on $\mathscr{E}$.*

Proof. Given an infinite sequence $e_0, e_1, \ldots \in \mathscr{E}$ there must be an infinite subsequence $e_{i_0}, e_{i_1}, \ldots$ such that $B(e_{i_0}) = B(e_{i_1}) = \ldots$. By Theorem 51,[13] there are k and l such that $e_{i_k} \trianglelefteq e_{i_l}$ and then $e_{i_k} \trianglelefteq^* e_{i_l}$, as required. □

[13] Recall that $\mathscr{E} = \mathscr{E}_{F \cup G \cup C}(V)$ where F, G, C are finite.

Theorem 60. *C terminates.*

Proof. The proofs of Propositions 50 and 57 can be repeated with $\trianglelefteq^*$ in place of $\trianglelefteq$. □

Theorem 61. *L terminates.*

Proof. Left as a challenging exercise. □

Martens and Gallagher show, essentially, that an abstract program transformer terminates if it maintains a predicate of the form in Proposition 43 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 38 and 43 (by Theorem 33).

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.

8. Concluding remarks

We have presented a framework for proving termination of program transformers and used it to prove termination of positive supercompilers and – very briefly – of a generic algorithm for partial deduction. It would be interesting to develop the latter proof into proofs of termination of other partial deduction algorithms. It would also be interesting to apply the framework to prove termination of other transformers, e.g. partial evaluators. These ideas are left for future work.

We hope to have demonstrated that termination proofs using our framework are independent of many of the particularities of the transformer, e.g. the language in which the programs to be transformed are written. Indeed, we have been able to develop results stating such properties as "it is fine to ignore the leaves of the tree in whatever test the transformer makes" (Proposition 46). Such results belong to a general theory of termination of apts, not to the development of one specific apt.

Instead of metric spaces we could have based our presentation on the less well-known *projection spaces* – see, e.g. [16]. Although these seem closer to the intuition behind our transformation trees, we have stuck to metric spaces since these are better known.

Acknowledgements

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, G. Vidal, Narrowing-driven partial evaluation of functional logic programs, in: H.R. Nielson (Ed.), European Symp. on Programming, Lecture Notes in Computer Science, Vol. 1058, Springer, Berlin, 1996, pp. 46–61.
[2] A. Arnold, M. Nivat, Metric interpretations of infinite trees and semantics of non deterministic recursive programs, Theoret. Comput. Sci. 11 (1980) 181–205.
[3] A. Arnold, M. Nivat, The metric space of infinite trees. Algebraic and topological properties Fund. Inform. III(4) (1980) 445–476.
[4] R. Bird, Tabulation techniques for recursive programs, ACM Comput. Surv. 12 (4) (1980) 403–417.
[5] D. Bjørner, A.P. Ershov, N.D. Jones (Eds.), 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, J. Comput. System Sci. 27 (1983) 225–255.
[7] S.L. Bloom, C.C. Elgot, J.B Wright, Vector iteration in pointed iterative theories, SIAM J. Comput. 9 (3) (1980) 525–540.
[8] S.L. Bloom, D. Patterson, Easy solutions are hard to find, in Colloquium on Trees in Algebra and Programming, Lecture Notes in Computer Science, vol. 112, Springer, Berlin, 1981, pp. 135–146.
[9] S.L. Bloom, R. Tindell, Compatible orderings on the metric theory of trees, SIAM J. Comput. 9 (4) (1980) 683–691.
[10] M. Bruynooghe, D. De Schreye, B. Krekels, Compiling control, J. Logic Programm., 6 (1989) 135–162.
[11] R.M. Burstall, J. Darlington, A transformation system for developing recursive programs, J. Assoc. Comput. Mach. 24 (1) (1977) 44–67.
[12] B. Courcelle, Fundamental properties of infinite trees, Theoret. Comput. Sci. 25 (1983) 95–169.
[13] O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, Lecture Notes in Computer Science, Vol. 1110, Springer, Berlin, 1996.
[14] N. Dershowitz, Termination of rewriting, J. Symbolic Comput. 3 (1987).
[15] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990, pp. 244–320.
[16] H. Ehrig, F. Parisi-Presicce, P. Boehm, C. Rieckoff, C. Dimitrovici, M. Große-Rohde, Combining data type and recursive process specifications using projection algebras, Theoret. Comput. Sci. 71 (1990) 347–380.
[17] A. Ferguson, P.L. Wadler, When will deforestation stop? in Glasgow Workshop on Functional Programming, 1988, pp. 39–56.
[18] Y. Futamura, K. Nogi, Generalized partial computation, in: D. Bjørner, A.P. Ershov, N.D. Jones (Eds.), Partial Evaluation and Mixed Computation, North-Holland, Amsterdam, 1988, pp. 133–151.
[19] R. Glück, J. Jørgensen, B. Martens, M.H. Sørensen, Controlling conjunctive partial deduction, in: H. Kuchen, D.S. Swierstra (Eds.), Programming Languages: Implementations, Logics and Programs, Lecture Notes in Computer Science, Vol. 1140, Springer, Berlin, 1996, pp. 137–151.
[20] R. Glück, M.H. Sørensen, Partial deduction and driving are equivalent, in: M. Hermenegildo, J. Penjam (Eds.), Programming Languages: Implementations, Logics and Programs, Lecture Notes in Computer Science, Vol. 844, Springer, Berlin, 1994, pp. 165–181.
[21] R. Glück, M.H. Sørensen, A roadmap to metacomputation by supercompilation, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, Lecture Notes in Computer Science, Vol. 1110, Springer, Berlin, 1996, pp. 137–160.
[22] G. Higman, Ordering by divisibility in abstract algebras, Proc. London Math. Soc. 3 (2) (1952) 326–336.
[23] N.D. Jones, The essence of program transformation by partial evaluation and driving, in: N.D. Jones, M. Hagiya, M. Sato (Eds.), Logic, Language, and Computation, Lecture Notes in Computer Science, Vol. 792, Springer, Berlin, 1994, pp. 206–224. Festschrift in honor of S. Takasu.
[24] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[25] J.B. Kruskal, Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture, Trans. Amer. Math. Soc. 95 (1960) 210–225.
[26] M. Leuschel, B. Martens, Global control for partial deduction through characteristic atoms and global trees, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, Lecture Notes in Computer Science, Vol. 1110, Springer, Berlin, 1996, pp. 263–283.

[27] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1984.
[28] M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Language Semantics, Lecture Notes in Computer Science, Vol. 298, Springer, Berlin, 1987.
[29] B. Martens, J. Gallagher, Ensuring global termination of partial deduction while allowing flexible polyvariance, in: L. Sterling (Ed.), Int. Conf. on Logic Programming, MIT Press, Cambridge, MA, 1995, pp. 597–613.
[30] J. Mycielski, W. Taylor, A compactification of the algebra of terms, Algebra Universalis 6 (1976) 159–163.
[31] C.St.J.A. Nash-Williams, On well-quasi-ordering finite trees, Proc. Cambridge Math. Soc. 59 (1963) 833–835.
[32] A. Pettorossi, A powerful strategy for deriving efficient programs by transformation, in ACM Conf. on Lisp and Functional Programming, ACM Press, New York, 1984, pp. 273–281.
[33] A. Pettorossi, M. Proietti, A comparative revisitation of some program transformation techniques, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, Lecture Notes in Computer Science, Vol. 1110, Springer, Berlin, 1996, pp. 355–385.
[34] M. Proietti, A. Pettorossi, The loop absorption and the generalization strategies for the development of logic programs and partial deduction, J. Logic Programm. 16 (1993) 123–161.
[35] W. Rudin, Principles of Mathematical Analysis, Mathematics Series, 3rd Edition, McGraw-Hill, New York, 1976.
[36] E. Ruf, D. Weise, On the specialization of online program specializers, J. Funct. Programm. 3 (3) (1993) 251–281.
[37] M.B. Smyth, Topology, in: S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (Eds.), Handbook of Logic in Computer Science, Vol. II, Oxford University Press, Oxford, 1992, pp. 641–761.
[38] M.H. Sørensen, R. Glück, An algorithm of generalization in positive supercompilation, in: J.W. Lloyd (Ed.), Logic Programming: Proc. 1995 Int. Symp., MIT Press, Cambridge, MA, 1995, pp. 465–479.
[39] M.H. Sørensen, R. Glück, N.D. Jones, Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation, in: D. Sannella (Ed.), European Symposium on Programming, Lecture Notes in Computer Science, Vol. 788, Springer, Berlin, 1994, pp. 485–500.
[40] M.H. Sørensen, R. Glück, N.D. Jones, A positive supercompiler, J. Funct. Programm. 6 (6) (1996) 811–838.
[41] M.H.B. Sørensen, Convergence of program transformers in the metric space of trees, in: J. Jeuring (Ed.), Mathematics of Program Construction, Lecture Notes in Computer Science, Vol. 1422, Springer, Berlin, 1998, pp. 315–337.
[42] H. Tamaki, T. Sato, Unfold/fold transformation of logic programs, in: S.-A. Tärnlund (Ed.), Int. Conf. on Logic Programming, Uppsala University, 1984, pp. 127–138.
[43] V.F. Turchin, The concept of a supercompiler, ACM Trans. Programm. Lang. Systems 8 (3) (1986) 292–325.
[44] V.F. Turchin, The algorithm of generalization in the supercompiler, in: D. Bjørner, A.P. Ershov, N.D. Jones (Eds.), Partial Evaluation and Mixed Computation, North-Holland, Amsterdam, 1988, pp. 531–549.
[45] V.F. Turchin, On generalization of lists and strings in supercompilation, Tech. Report CSc. TR 96-002, City College of the City University of New York, 1996.
[46] P.L. Wadler, Deforestation: transforming programs to eliminate intermediate trees, Theoret. Comput. Sci. 73 (1990) 231–248.
[47] D. Weise, R. Conybeare, E. Ruf, S. Seligman, Automatic online partial evaluation, in: J. Hughes (Ed.), Conf. on Functional Programming and Computer Architecture, Lecture Notes in Computer Science, Vol. 523, Springer, Berlin, 1991, pp. 165–191.