An Algorithm of Generalization in Positive Supercompilation

Morten H. Sørensen & **Robert Glück**
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
{rambo,glueck}@diku.dk

Abstract

This paper presents a termination technique for positive supercompilation, based on notions from term algebra. The technique is not particularily biased towards positive supercompilation, but also works for deforestation and partial evaluation. It appears to be well suited for partial deduction too. The technique guarantees termination, yet it is not overly conservative.

Our technique can be viewed as an instance of Martens' and Gallagher's recent framework for global termination of partial deduction, but it is more general in some important respects, *e.g.* it uses well-quasi orderings rather than well-founded orderings. Its merits are illustrated on several examples.

1 Introduction

Program specialization has received much attention in the area of logic and functional languages, *e.g. partial deduction* [12, 11, 4], *partial evaluation* [9], and *deforestation* [26]. *Supercompilation* [23] is a program transformation technique for functional languages that is strictly stronger than partial evaluation and deforestation. It is capable of theorem proving and program inversion [20, 21, 22], and of program optimization beyond partial evaluation and deforestation [5]. Recent renewed interest in supercompilation has lead to the *positive supercompiler* [6, 18, 19], a simplified version of Turchin's supercompiler for a functional language with trees as data structures.

Supercompilation consists of *driving* and *generalization*, a technique to ensure termination of driving. A termination technique for the original supercompiler exists [24]. The present paper gives a termination technique for the positive supercompiler, solidly founded on techniques from term algebra. It grew out of a characterization of non-termination patterns [18, Ch.12] and our aim to formalize more generally a technique extending [24] implemented in the latest version of Turchin's supercompiler.

Several termination techniques exist for partial evaluation and deforestation. However, program specialization based on *constant propagation*, such as conventional partial evaluation, and *unification-based* methods, such as supercompilation and partial deduction, have quite different termination problems. The latter techniques are more aggressive, more *perfect* in the terminology of [6], so ensuring termination is harder.

The authors have established a correspondence between driving and partial deduction [8]. Indeed, our technique can be viewed as an instance of Martens' and Gallagher's recent framework [14] for ensuring global termination of partial deduction. However, our technique is more general in some important respects, *e.g.* in using *well-quasi orderings*, rather than well-founded orderings, and appears to be well suited for partial deduction.

Section 2 describes the object language. Section 3 introduces driving and Section 4 defines our termination technique. Section 5 gives examples, Section 6 considers partial deduction, and Section 7 describes related work.

2 Object Language

Our setting is a first-order functional language with a normal-order (lazy) semantics, similar to a first-order fragment of Miranda.

Definition 2.1 (language syntax.) We assume denumerable, disjoint sets of function symbols $f \in \mathcal{F}$, constructor symbols $c \in \mathcal{C}$, and variable symbols $x \in \mathcal{X}$. All symbols are applied to a fixed number of arguments corresponding to the arity of the symbol ($n, m \geq 0, k > 0$).

$$\begin{array}{lcll}
\mathcal{Q} \ni q & ::= & f\, x_1 \ldots x_n = t \quad \cdots \quad g\, y_1 \ldots y_m = s & \text{(program)} \\[4pt]
\mathcal{T} \ni s, t & ::= & x & \text{(variable)} \\
 & \mid & c\, t_1 \ldots t_n & \text{(constructor)} \\
 & \mid & f\, t_1 \ldots t_n & \text{(function call)} \\
 & \mid & \textbf{case}\ t_0\ \textbf{of}\ \ p_1 \to t_1; \ldots; p_k \to t_k & \text{(case-expression)} \\
 & \mid & \textbf{if}\ t_1 = t_2\ \textbf{then}\ t_3\ \textbf{else}\ t_4 & \text{(conditional w/equality test)} \\[4pt]
\mathcal{P} \ni p & ::= & c\, x_1 \ldots x_n & \text{(flat pattern)}
\end{array}$$

The variables in the patterns of a case-expression are *bound*; all others variables are *free*. We write $t \equiv t'$ if t and t' differ only in the names of bound variables.

We require that each function in a program has exactly one definition and that all free variables in the right side of a definition be present in its left side. As usual, we require that all patterns in a case-expression have different constructors, and that no pattern contain the same variable twice.

Example 2.2 For the list constructors *nil* and *cons x xs* we use the usual notation $[\,]$ and $(x : xs)$. The *remove* function f replaces each pair of identical symbols in a list by a single symbol [24]:

```
f x  =  case x of
            (a : t) → case t of
                          (b : s) → if a=b then a : (f s) else a : (f t)
                           []     → a : []
             []     → []
```

3 Driving

Driving takes a term, possibly containing free variables, and a program and constructs a possibly infinite *process tree*. Each node contains a term t and its children contain the terms that arise by one *normal-order reduction* step from t. Whenever the reduction step has different possible outcomes, for instance if the term is a case test on a variable, there will be several children so as to account for all possibilities.

Subsection 3.1 introduces some notions used to define normal-order reduction in Subsection 3.2. Subsection 3.3 describes process trees.

3.1 Redex and Evaluation Context

Definition 3.1 (value, observable, redex, evaluation context.)

$$
\begin{array}{rcll}
\mathcal{B} \ni b & ::= & x \mid c\, b_1 \ldots b_n & \text{(value)} \\
\mathcal{O} \ni o & ::= & x \mid c\, t_1 \ldots t_n & \text{(observable)} \\
\\
\mathcal{R} \ni r & ::= & f\, t_1 \ldots t_n & \text{(redex)} \\
 & \mid & \mathbf{case}\ o\ \mathbf{of}\ p_1 \!\to\! t_1; \ldots; p_m \!\to\! t_m & \\
 & \mid & \mathbf{if}\ b_1 = b_2\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 & \\
\\
\mathcal{E} \ni e & ::= & [\!] & \text{(evaluation context)} \\
 & \mid & \mathbf{case}\ e\ \mathbf{of}\ p_1 \!\to\! t_1; \ldots; p_m \!\to\! t_m & \\
 & \mid & \mathbf{if}\ d = t_2\ \mathbf{then}\ t_3\ \mathbf{else}\ t_4 & \\
 & \mid & \mathbf{if}\ b = d\ \mathbf{then}\ t_3\ \mathbf{else}\ t_4 & \\
d & ::= & e \mid c\, b_1 \ldots b_{i-1}\, d\, t_{i+1} \ldots t_n &
\end{array}
$$

The expression $e[t]$ denotes the result of replacing the 'hole' $[\!]$ in e by t.

A *value* is a term built exclusively from constructors and variables. A value is *ground* if it contains no variables. An *observable* is a variable or a term with an outermost constructor. A term which is not an observable has the form $e[r]$ where the *redex* r is the outermost reducible subterm and the *evaluation context* e is the surrounding part of the term.

Lemma 3.2 (the unique decomposition property.) For any $t \in \mathcal{T}$ either $t \in \mathcal{O}$ or there exists a unique pair $(e, r) \in \mathcal{E} \times \mathcal{R}$ such that $t \equiv e[r]$.

Example 3.3 In the term $f\ (g\ t)$ the redex is the whole term, and the context is empty (*i.e.* $[\!]$). So normal-order reduction unfolds the call to f before g. In the term **case** $(f\ t)$ **of** $p_1 \!\to\! t_1; \ldots; p_m \!\to\! t_m$ the redex is $f\ t$ and the context is **case** $[\!]$ **of** $p_1 \!\to\! t_1; \ldots; p_m \!\to\! t_m$. That is, normal-order reduction unfolds the call to f; the case-expression cannot be unfolded because the tested term does not have an outermost constructor.

3.2 Normal-Order Reduction

The rules for normal-order reduction are given by the map $\mathcal{N}$ from terms to ordered sequences $\langle t_1, \ldots, t_n \rangle$ of terms. The clauses of $\mathcal{N}$ are mutually exclusive and together exhaustive by the unique decomposition property.

Definition 3.4 (normal-order reduction step.)

t	$\mathcal{N}[\![t]\!]$	
x	$\langle\,\rangle$	
$c\,t_1\ldots t_n$	$\langle t_1,\ldots,t_n\rangle$	
$e[f\,t_1\ldots t_n]$	$\langle e[t\{x_1 := t_1\ldots x_n := t_n\}]\rangle$	if $f\,x_1\ldots x_n = t$
$e[$**case** $(c\,t_1\ldots t_n)$ **of** $p_1\to s_1;\ldots;p_m\to s_m]$	$\langle e[s_j\{x_1 := t_1\ldots x_n := t_n\}]\rangle$	if $p_j \equiv c\,x_1\ldots x_n$
$e[$**case** x **of** $p_1\to s_1;\ldots;p_m\to s_m]$	$\langle (e[s_1])\{x := p_1\},\ldots,(e[s_m])\{x := p_m\}\rangle$	
$e[$**if** $b{=}b'$ **then** t **else** $t']$	$\langle e[t]\rangle$	if b, b' are ground, $b \equiv b'$
	$\langle e[t']\rangle$	if b, b' are ground, $b \not\equiv b'$
	$\langle (e[t])\lceil b, b'\rceil, e[t']\rangle$	if b, b' are not both ground

We use the notation $t\{x_1 := t_1,\ldots,x_n := t_n\}$ for the simultaneous substitution of $t_1,\ldots,t_n$ for the free occurrences of $x_1,\ldots,x_n$, respectively, in t. $\lceil b, b'\rceil$ denotes an idempotent most general unifier $\{x_1 := t_1,\ldots,x_n := t_n\}$ of b and b' if it exists, and *fail* otherwise, where we stipulate $t\,\mathit{fail} \equiv t$.

In the third clause the bound variables in the definition of f should be renamed in order to avoid name clashes.

Note the substitutions in the clauses for case tests on variables and equality tests on non-ground terms. The assumed outcome of the test is propagated to the terms resulting from the step. We call this *unification-based* information propagation.

3.3 Process Trees

A *marked process tree* is a directed acyclic graph where each node is labeled with a term t and all edges leaving a node are ordered. One node is chosen as the *root*, and every leaf node may have an additional *mark*.

Driving is the action of constructing process trees, and is characterized by two essential principles: *normal-order reduction* (as opposed to applicative order typical in partial evaluation) and *unification-based information propagation* (as opposed to constant propagation typical in partial evaluation and deforestation). For a more detailed comparison see [6, 19, 15].

Algorithm 3.5 (driving.)

```
1. INPUT $t_0 \in \mathcal{T}$, $q \in \mathcal{Q}$
2. LET $T$ be the marked process tree with unmarked node labeled $t_0$.
3. WHILE there exists an unmarked leaf node $N$ in $T$ with label $t$:
   (a) LET $\mathcal{N}[\![ t ]\!] = \langle t_1,\ldots,t_n\rangle$.
   (b) IF $n = 0$ THEN mark $N$
       ELSE add $n$ children to $N$ and label them $t_1, \dots, t_n$

4. OUTPUT $T$
```

4 The Termination Strategy

In the previous section we introduced driving to construct a potentially infinite process tree representing all computations of a given term in a given program. We now define a termination strategy to construct instead a finite graph, which we call a *partial process tree.*

The basic idea is as follows: if a leaf node M has an ancestor L and it "seems likely" that continued driving will generate an infinite sequence $L, \dots, M, N, \dots$ then M should not be driven any further. In Subsection 4.2 we define a criteria, a so-called *whistle*, that formalizes this decision.

The intention with a finite partial process tree is that it represents a program, where a node M corresponds to a function f_M and an edge from M to N corresponds to a call from f_M to f_N. However, for a finite partial process tree to have this property it is not always sufficient to stop driving at a leaf M with an ancestor L that possibly leads to an infinite branch. If M is an instance of L, a return ("dashed") edge can be added from M to L corresponding to a recursive call. This action is called *folding.* Otherwise the action taken will be to replace the whole subtree with root L by a new node G whose label is a *generalization* of the labels of L, M. Any edges previously into L, now going into K, still represent calls, and due to the generalization it is now "less likely" that an infinite branch will begin in K.

Subsection 4.3 defines generalization, and Subsection 4.4 defines the complete *positive supercompiler* based on whistling, folding and generalization.

4.1 Another View of Terms

Definition 4.1 (uniform terms.) We assume denumerable, disjoint sets of variables $\mathcal{X}$ and fixed-arity operator symbols $\sigma \in \Sigma$. The set of terms $T_\Sigma(\mathcal{X})$ is the smallest set containing $\mathcal{X}$ such that $\sigma(t_1, \dots, t_n) \in T_\Sigma(\mathcal{X})$ whenever $t_1, \dots, t_n \in T_\Sigma(\mathcal{X})$ and n is the arity of σ.

One can view all terms of the object language in Definition 2.1 as special case of this definition, where $\Sigma = \mathcal{F} \cup \mathcal{C} \cup \{\mathbf{ifthenelse}, \mathbf{caseof}\}$. The **caseof** construction entails a bit of difficulty since the p_i's in it are not general terms but patterns binding variables. Nevertheless, for elegance we state below some definitions and results for terms from Definition 4.1, and treat the translation to terms from Definition 2.1 informally.

4.2 When to Stop?

We stop driving at a leaf node with label t if one of its ancestors has label s and $s \trianglelefteq t$, where $\trianglelefteq$ is the *homeomorphic embedding relation* known from term algebra [3]. Variants of this relation are used in termination proofs for term rewrite systems [3] and for ensuring local termination of partial deduction [1].

The rationale behind using this relation is that, on the one hand, in any infinite sequence $t_0, t_1, \ldots$ there *definitely* exists some $i < j$ with $t_i \trianglelefteq t_j$. This ensures that driving does not proceed infinitely. On the other hand, if $t_i \trianglelefteq t_j$ then all the subterms of t_i are present in t_j embedded in extra subterms. This suggests that t_j might arise from t_i by some infinitely continuing system and that driving will be stopped for a good reason.

Definition 4.2 (homeomorphic embedding relation.) Define $\trianglelefteq$ as the smallest relation on $T_\Sigma(\mathcal{X})$ satisfying:

Variable	*Diving*	*Coupling*
$x \trianglelefteq y$	$\dfrac{s \trianglelefteq t_i \text{ for some } i}{s \trianglelefteq \sigma\,(t_1, \ldots, t_n)}$	$\dfrac{s_1 \trianglelefteq t_1, \ldots, s_n \trianglelefteq t_n}{\sigma\,(s_1, \ldots, s_n) \trianglelefteq \sigma\,(t_1, \ldots, t_n)}$

Based on this definition it is not hard to give an algorithm *WHISTLE*(M, N) deciding whether $M \trianglelefteq N$.

Note that diving detects a subterm embedded in a larger term, and coupling matches the subterms of two terms. The corresponding rules for $\mathcal{T}(\mathcal{X}, \mathcal{F})$ should be clear, except perhaps the coupling rule for **case**:

$$\frac{s_i \trianglelefteq t_i \text{ for all } i, \text{ and } p_i \text{ is a renaming of } q_i \text{ for all } i}{\textbf{case } s_0 \textbf{ of } p_1 \rightarrow s_1; \ldots; p_n \rightarrow s_n \trianglelefteq \textbf{case } t_0 \textbf{ of } q_1 \rightarrow t_1; \ldots; q_n \rightarrow t_n}$$

Example 4.3

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

Definition 4.4 (well-founded order, well-quasi order.)

1. A *well-founded order* (wfo) on a set S is an anti-reflexive, transitive, anti-symmetric relation $<_S$ on S such that there are no infinite descending sequences $s_1 >_S s_2 >_S \ldots$ of elements from S.

2. A *well-quasi order* (wqo) on a set S is a reflexive, transitive relation $\leq_S$ on S such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \leq_S s_j$.

4.3 How to Generalize?

We define the generalization of two terms t_1, t_2 as the most specific generalization $\lfloor t_1, t_2 \rfloor$, known from term algebra [3]. The same operation has similar use in partial deduction, see *e.g.* [14].

The rationale behind using this definition is that when one generalizes two terms one replaces parts of the terms by variables. This, in effect, implies a loss of knowledge about the term. The most specific generalization operation entails the smallest possible loss of knowledge. With respect to termination of the supercompiler the essential property of $\lfloor \bullet, \bullet \rfloor$ is that there is a wfo $>$ such that $t_1, t_2 > \lfloor t_1, t_2 \rfloor$ whenever we perform a generalization step in the supercompiler.

Definition 4.5 (instance, generalization, msg.) Given $t_1, t_2 \in T_\Sigma(\mathcal{X})$.

1. An *instance* of t_1 is a term of the form $t_1\theta$ where θ is a substitution.
2. A *generalization* of t_1, t_2 is a triple $(t_g, \theta_1, \theta_2)$ where θ_1, θ_2 are substitutions such that $t_g\theta_1 \equiv t_1$ and $t_g\theta_2 \equiv t_2$.
3. A *most specific generalization* (msg) of t_1 and t_2 is a generalization $(t_g, \theta_1, \theta_2)$ of t_1 and t_2 such that for every other generalization $(t'_g, \theta'_1, \theta'_2)$ of t_1 and t_2 it holds that t_g is an instance of t'_g.

The following algorithm computes $\lfloor \bullet, \bullet \rfloor$ for the set of terms $T_\Sigma(\mathcal{X})$.

Algorithm 4.6 (msg.) The most specific generalization $\lfloor s, t \rfloor$ of two terms $s, t \in T_\Sigma(\mathcal{X})$ is computed by exhaustively applying the following rewrite rules to the initial triple $(x, \{x := s\}, \{x := t\})$:

$$\begin{pmatrix} t_g & & \\ \{x := \sigma(s_1, \ldots, s_n)\} & \cup & \theta_1 \\ \{x := \sigma(t_1, \ldots, t_n)\} & \cup & \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x := \sigma(y_1, \ldots, y_n)\} & & \\ \{y_1 := s_1, \ldots, y_n := s_n\} & \cup & \theta_1 \\ \{y_1 := t_1, \ldots, y_n := t_n\} & \cup & \theta_2 \end{pmatrix}$$

$$\begin{pmatrix} t_g & & \\ \{x := s, y := s\} & \cup & \theta_1 \\ \{x := t, y := t\} & \cup & \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x := y\} & & \\ \{y := s\} & \cup & \theta_1 \\ \{y := t\} & \cup & \theta_2 \end{pmatrix}$$

Example 4.7

s		t	t_g	θ_1	θ_2
b	$\unlhd$	$a(b)$	x	$\{x := b\}$	$\{x := a(b)\}$
$c(b)$	$\unlhd$	$c(a(b))$	$c(x)$	$\{x := b\}$	$\{x := a(b)\}$
$c(y)$	$\unlhd$	$c(a(y))$	$c(x)$	$\{x := y\}$	$\{x := a(y)\}$
$d(b, b)$	$\unlhd$	$d(a(b), a(b))$	$d(x, x)$	$\{x := a(b)\}$	$\{x := a(b)\}$

The corresponding notion of generalization for terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is slightly more complicated due to bound variables in case tests.

The following operation will needed in connection with generalizations.

Definition 4.8 (split.) For $t \in T_\Sigma(\mathcal{X}) \backslash \mathcal{X}$ define *split*(t) by:

$$split(\sigma(t_1, \ldots, t_n)) = (\sigma, \langle t_1, \ldots, t_n \rangle)$$

4.4 Partial Process Trees

A *marked partial process tree* is a marked process tree except that there are two extra kinds of nodes: fold- and generalization-nodes.

- An *F-node* has a label $F\ x_1 \ldots x_n$ and n children together with a dashed edge to an ancestor ($n \geq 0$).
- A *G-node* has a label $G\ x_1 \ldots x_n$ and $n+1$ children ($n \geq 0$).

An *F-node* N in a partial process tree reflects the fact that when the node was to be driven it was decided that a fold steps should occur. The dashed edge goes to the node representing the function to be called recursively, and the children of N are the arguments in this call. A *G-node* M with label $G\, x_1, \ldots, x_n$ reflects the fact that a generalization of two nodes with labels s, t was made. The label of the first child of M is the generalized term t_g and the n children have labels $t_1, \ldots, t_n$ such that $s \equiv t_g\{x_1 := t_1, \ldots, x_n := t_n\}$.

We will still regard a partial process tree as an acyclic graph by ignoring dashed edges. The terms *ancestor, leaf, etc.* apply only to non-dashed edges. The labels on F- and G-nodes are unrelated to all other labels wrt. $\trianglelefteq$.

Definition 4.9 Let T be a marked partial process tree and N an unmarked leaf node with label t. We define three operations that yield a new marked partial process tree.

1. *UNFOLD*(T, N) is the tree obtained as follows. Let $\mathcal{N}[\![t]\!] = \langle t_1, \ldots, t_n \rangle$. If $n = 0$ mark N, otherwise add n unmarked children labeled $t_1, \ldots, t_n$.

2. Suppose that N has an ancestor M with label s where t is an instance of s, *i.e.* $t \equiv s\{x_1 := t_1 \ldots x_n := t_n\}$. *FOLD*$(T, M, N)$ is the tree obtained as follows. Replace the label of N with the label $F\ x_1 \ldots x_n$ and add a dashed edge to M. If $n = 0$ then mark N, otherwise add n unmarked children to N with labels $t_1, \ldots, t_n$.

3. Suppose that N has an ancestor M with label s such that $\lfloor s, t \rfloor = (t_g, \{x_1 := t_1, \ldots, x_n := t_n\}, \theta)$. *GENERALIZE*$(T, M, N)$ is the tree obtained as follows.

   (a) If t_g is not a variable: delete all descendants of M, give M label $G\ x_1 \ldots x_n$ and let M have $m+1$ unmarked children labeled $t_g, t_1, \ldots, t_m$. Dashed edges from M or its decendents are erased.

   (b) If t_g is a variable: let *split*$(t) = (\sigma, \langle t_1, \ldots, t_m \rangle)$ and let $s_g \equiv \sigma(y_1, \ldots, y_m)$ where $y_1, \ldots, y_m$ are new variables, and give N label $G\ y_1 \ldots y_m$ and $m+1$ unmarked children with labels $s_g, t_1, \ldots, t_m$.

Algorithm 4.10 (positive supercompilation.)

1. INPUT $t_0 \in \mathcal{T}$, $q \in \mathcal{Q}$
2. LET T_0 be the marked partial process tree with one unmarked node labeled t_0. SET $i := 0$.

```
3. WHILE there exists an unmarked leaf node $N$ in $T_i$:
   (a) IF there exists no ancestor $M$ such that $\mathit{WHISTLE}(M, N)$
       THEN $T_{i+1} := \mathit{UNFOLD}(T_i, N)$
       ELSE
         i. LET $M$ be the nearest ancestor such that $\mathit{WHISTLE}(M, N)$
        ii. IF node $N$ is an instance of $M$
            THEN $T_{i+1} := \mathit{FOLD}(T_i, M, N)$
            ELSE $T_{i+1} := \mathit{GENERALIZE}(T_i, M, N)$
   (b) SET $i := i + 1$
4. OUTPUT $T_{i-1}$
```

This formulation of the supercompilation algorithm is in accordance with Turchin's *generalization principle* [24] which states that a generalization between two terms has a meaning only if they have a common computation history. Indeed, our algorithm searches only the ancestors of a leaf node.

Note that our algorithm does not specify a particular strategy for *selecting unmarked leaf nodes.* In particular, one may chose a breath-first or depth-first strategy. Variants of the algorithm may employ different strategies for *selecting ancestors* for folding and generalization. For instance, one may, instead of chosing the closest ancestor, select the ancestor that gives the most specific generalization (*e.g.* if $s_1 \trianglelefteq t$ and $s_2 \trianglelefteq t$ for two ancestors with labels s_1, s_2 then there is a choice between $\lfloor s_1, t \rfloor$ and $\lfloor s_2, t \rfloor$).

The following theorem is proved in an extended version of this paper. The crucial properties is that $\trianglelefteq$ is a wqo and that there is a wfo such that whenever *GENERALIZE* is used, the new term is strictly smaller according to this wfo.

Theorem 4.11 *Algorithm 4.10 always terminates.*

5 Examples

In this section we illustrate how our technique is able to handle several situations that typically occur in constant propagation (Subsection 5.2), deforestation (Subsection 5.3), and unification-based information propagation (Subsection 5.4).

5.1 Transient reductions

We adopt a slight modification of our technique in order to shorten the examples and improve the results. Terms of the following three forms are called *transient*.

$$\begin{array}{l} e[\mathbf{case}\ (c\, t_1 \ldots t_n)\ \mathbf{of}\ \ p_1 \rightarrow s_1; \ldots; p_m \rightarrow t_m] \\ e[f\, t_1 \ldots t_n] \\ e[\mathbf{if}\ b{=}b'\ \mathbf{then}\, t\ \mathbf{else}\, t'] \text{ with } b, b' \text{ ground} \end{array}$$

Following [23] we proceed as follows. In every iteration of the while loop in Algorithm 4.10 with a transient term we perform an unfold step, a *transient reduction*, without checking ancestors for whistling. Moreover, for the remaining types of terms, when we compare to ancestors, we only compare to ancestors that are not transient.

The rationale behind this is that any loop in the program must pass through a non-transient term unless there is an unconditional loop in the program. However, this means that the partial process tree in principle can be infinite—a risk considered worth taking in the area of partial evaluation [9]. Transient terms are similar to *determinate goals* in partial deduction, and "unfolding determinate goals is almost always a good idea" [4, p92].

5.2 Diving and Coupling

Functions that recursively build a growing stack of function calls are a typical situation encountered during driving. Such recursive calls may either occur on the "top level" or be embedded in a common context. We show how our algorithm works in both cases.

Consider the recursive definition of the reverse function:

$$\begin{array}{lcll} f\ x & = & \mathbf{case}\ x\ \mathbf{of}\ \ (z : zs)\to g\ (f\ zs)\ z;\ \ []\to[] & \equiv\ \ \mathbf{case}^1(x) \\ g\ x\ y & = & \mathbf{case}\ x\ \mathbf{of}\ \ (z : zs)\to z : (g\ zs\ y);\ \ []\to[y] & \equiv\ \ \mathbf{case}^2(x, y) \end{array}$$

where we have introduced short hands $\mathbf{case}^1(x)$ and $\mathbf{case}^2(x, y)$ for the right hand sides of f and g, respectively. We will also allow terms in place of x, y. With the initial term $f\ u$, a stage of the partial process tree, showing only the non-transient terms, is:

$$\begin{tikzcd} & \boxed{\mathbf{case}^1(u)} \arrow[dl] \arrow[dr] & \\ \boxed{\mathbf{case}^2(\mathbf{case}^1(zs), z)} & & \boxed{[]} \end{tikzcd}$$

Here the lower leftmost term and the term at the root must be generalized, since $\mathbf{case}^1(u) \trianglelefteq \mathbf{case}^2(\mathbf{case}^1(zs), z)$ by the diving rule. This is sensible since otherwise the left-most branch would continue *ad infinitum*. The *prefix* criterion in Turchin's generalization algorithm [24] has the same effect of stopping this development.

Developing further we arrive at the following finite partial process tree:

$$\begin{tikzcd} & & & & \boxed{\mathbf{case}^1(u)} \arrow[dl] \arrow[dr] & \\ & & & \boxed{G\ v} \arrow[dl] \arrow[dr] & & \boxed{[]} \\ & & \boxed{\mathbf{case}^2(v, z)} \arrow[dl] \arrow[dr] & & \boxed{\mathbf{case}^1(zs)} \arrow[uu, dashed] & \\ & \boxed{z' : g\ zs'\ z} \arrow[dl] \arrow[dr] & & \boxed{[z]} & & \\ \boxed{z'} & & \boxed{\mathbf{case}^2(zs', z)} \arrow[uu, dashed] & & & \end{tikzcd}$$

A term and program are constructed by introducing for every non-leaf node N a function whose right hand side is made up of the labels of the children of N, and whose formal parameters are the free variables in N's label:

$$\begin{array}{lcl} f_1\ u & = & \mathbf{case}\ u\ \mathbf{of}\ \ (z:zs)\rightarrow f_2\ z\ zs;\ \ []\rightarrow[] \\ f_2\ z\ zs & = & f_3\ (f_1\ zs)\ z \\ f_3\ v\ z & = & \mathbf{case}\ v\ \mathbf{of}\ \ (z':zs')\rightarrow f_4\ z'\ zs'\ z;\ \ []\rightarrow[z] \\ f_4\ z'\ zs'\ z & = & z' : f_3\ zs'\ z \end{array}$$

The term is $f_1\ u$. This is essentially the original term and program, the desired result. In the remainder we will only show parts of partial process trees, and omit construction of new terms and programs. More details appear in [8].

Now let the initial term be $\mathbf{case}^3(f\ u)$ with the further abbreviation

$$\mathbf{case}^3(x) \ \equiv\ \mathbf{case}\ s\ \mathbf{of}\ \ (z:zs)\rightarrow zs;\ \ []\rightarrow[]$$

The two first transient terms in the left-most branch of the process tree are:

$$\begin{array}{ll} (1) & \mathbf{case}^3(\mathbf{case}^1(u)) \\ (2) & \mathbf{case}^3(\mathbf{case}^2(\mathbf{case}^1(zs), z)) \end{array}$$

Again $(1) \trianglelefteq (2)$, but this time it follows using first coupling and then diving. $\lfloor(1),(2)\rfloor$ splits configuration (1) into two parts: $\mathbf{case}^3(v)$ and $\mathbf{case}^1(u)$.

5.3 Coupling

Recall the *remove* function f defined in Section 1. An alternative definition is $f\ u = \mathbf{case}^1(u)$ where we use abbreviations:

$$\begin{array}{lcl} \mathbf{case}^1(u) & \equiv & \mathbf{case}\ u\ \mathbf{of}\ \ (a:t)\rightarrow \mathbf{case}^2(a,t);\ \ []\rightarrow[] \\ \mathbf{case}^2(a,t) & \equiv & \mathbf{case}\ t\ \mathbf{of}\ \ (b:s)\rightarrow \mathbf{if}^3(a,b,t,s);\ \ []\rightarrow[a] \\ \mathbf{if}^3(a,b,t,s) & \equiv & \mathbf{if}\ a{=}b\ \mathbf{then}\ a:(f\ s)\ \mathbf{else}\ a:(f\ t) \end{array}$$

With initial term $f\ (f\ (f\ u))$ the left-most branch of the process tree is:

$$\begin{array}{llll} (1) & \mathbf{case}^1(\mathbf{case}^1(\mathbf{case}^1(u))) & (9) & \mathbf{case}^2(a, \mathbf{case}^1(\mathbf{case}^2(a_2,t_2))) \\ (2) & \mathbf{case}^1(\mathbf{case}^1(\mathbf{case}^2(a,t))) & (10) & \mathbf{case}^2(a, \mathbf{case}^1(\mathbf{if}^3(a_2,b_2,t_2,s_2))) \\ (3) & \mathbf{case}^1(\mathbf{case}^1(\mathbf{if}^3(a,b,t,s))) & (11) & \mathbf{case}^2(a, \mathbf{case}^2(a_2, \mathbf{case}^1(s_2))) \\ (4) & \mathbf{case}^1(\mathbf{case}^2(a, \mathbf{case}^1(s))) & (12) & \mathbf{case}^2(a, \mathbf{case}^2(a_2, \mathbf{case}^2(a_3,t_3))) \\ (5) & \mathbf{case}^1(\mathbf{case}^2(a, \mathbf{case}^2(a_1,t_1))) & (13) & \mathbf{case}^2(a, \mathbf{case}^2(a_2, \mathbf{if}^3(a_3,b_3,t_3,s_3))) \\ (6) & \mathbf{case}^1(\mathbf{case}^2(a, \mathbf{if}^3(a_1,b_1,t_1,s_1))) & (14) & \mathbf{case}^2(a, \mathbf{if}^3(a_2,a_3,a_3:(f\ s_3), f\ s_3)) \\ (7) & \mathbf{case}^1(\mathbf{if}^3(a,a_1,a_1:(f\ s_1), f\ s_1)) & (15) & \mathbf{if}^3(a,a_2,a_2:(f\ (f\ s_3)), f\ (f\ s_3)) \\ (8) & \mathbf{case}^2(a, \mathbf{case}^1(\mathbf{case}^1(s_1))) & (16) & \mathbf{case}^1(\mathbf{case}^1(\mathbf{case}^1(s_3))) \end{array}$$

Note that the development is not stopped prematurely although some terms are followed by larger terms, while other terms differ only in the order of constructors, such as (2), (4) and (8). The result is a program in which all three passes of the initial term are fused into a single pass. This example also illustrates the deforestation effect of supercompilation.

Turchin's generalization [24] achieves the same effect, but requires a more sophisticated strategy for structuring the function stack and "rewriting the computation history".

5.4 Pattern Matching

Given a *pattern* $p = [p_1, \ldots, p_m]$ and a *string* $s = [s_1, \ldots, s_n]$ the *matching problem* is to decide whether $s = [s_1, \ldots, s_i, p_1, \ldots, p_m, s_{m+i+1}, \ldots, s_n]$. The following *general pattern matcher* carries out this task.

$$\begin{array}{lcl}
match\ p\ s & = & loop\ p\ s\ p\ s \\
loop\ pp\ ss\ op\ os & = & \mathbf{case}^1(pp, ss, op, os) \\
\mathbf{case}^1(pp, ss, op, os) & \equiv & \mathbf{case}\ pp\ \mathbf{of}\ \ [\,]{\to}T;\ (p : pp){\to}\mathbf{case}^2(p', pp', ss, op, os) \\
\mathbf{case}^2(p', pp', ss, op, os) & \equiv & \mathbf{case}\ ss\ \mathbf{of}\ \ [\,]{\to}F;\ (s : ss){\to}\mathbf{if}^3(p', pp', s', ss', op, os) \\
\mathbf{if}^3(p', pp', s', ss', op, os) & \equiv & \mathbf{if}\ p{=}s\ \mathbf{then}\ loop\ pp\ ss\ op\ os\ \mathbf{else}\ \mathbf{case}^4(op, os) \\
\mathbf{case}^4(op, os) & \equiv & \mathbf{case}\ os\ \mathbf{of}\ \ [\,]{\to}F;\ (o : os){\to}loop\ op\ os\ op\ os
\end{array}$$

Suppose a pattern $[p_1, \ldots, p_m]$ is given and we want to solve the pattern matching problem for a number of strings. Then one can get a more efficient program by taking into account that the elements $[p_1, \ldots, p_n]$ are known. If the pattern is $[A, A, B]$ and the string has form $[A, A, s, \ldots]$ where s is not an A then after reading the two A's the specialized matcher may proceed by comparing the second A of the pattern to s.

Building the partial process tree for, *e.g.* the term $match\,[A, A, B]\,s$ and subsequently generating a new term and program gives a more effcient program with the same complexity as the specialized matchers produced by the well-known Kuth-Morris-Pratt algorithm, as shown in [18]. This is a classical problem in program transformation; for references see [19].

Neither partial evaluation nor deforestation can achieve this transformation, as explained in [19]. Partial deduction achieves the same effect in the logic programming context [4, 17] as driving in the functional setting [5]. This is not surprising given the close relation of driving and partial deduction [8].

Here is one branch of the finite partial process tree:

$$\begin{array}{ll}
(1) & \mathbf{case}^2(A, [A, B], ss, [A, A, B], ss) \\
(2) & \mathbf{if}^3(A, [A, B], s_1, ss_1, [A, A, B], (s_1 : ss_1)) \\
(3) & \mathbf{case}^2(A, [B], ss_1, [A, A, B], (A : ss_1)) \\
(4) & \mathbf{if}^3(A, [B], s_2, ss_2, [A, A, B], (A : s_2 : ss_2)) \\
(5) & \mathbf{case}^2(A, [\,], ss_2, [A, A, B], (A : A : ss_2)) \\
(6) & \mathbf{if}^3(A, [\,], s_3, ss_3, [A, A, B], (A : A : s_3 : ss_3)) \\
(7) & \mathbf{if}^3(A, [B], s_3, ss_3, [A, A, B], (A : s_3 : ss_3))
\end{array}$$

Note that the development of (4) is not stopped prematurely since $(2) \ntrianglelefteq (4)$, but continues until $(4) \trianglelefteq (7)$ which is essential to achieve the KMP-effect.

6 Application to Partial Deduction

In this section we briefly review the connection between the present technique and the framework of Martens and Gallagher [14] for ensuring global termination of partial deduction. We assume that the reader is familar with partial deduction as introduced in the papers [12, 11, 4].

First of all, as is noted in [14], we do not distinguish between local and global termination in supercompilation. Rather we perform always exactly one step of unfolding. We do not perceive this as being conservative; rather we push the problem of ensuring local termination to the global level.

Second, the central aspects of the Martens-Gallagher framework is that one chooses a wfo on the set of all atoms; or rather one partitions the set of atoms into k disjoint classes and chooses a well-founded order on each. Moreover, one also chooses an *abstraction operator* on atoms; the canonical example is the *most specific generalization* seen above. One then develops partial SLD trees similar to how we develop partial process trees above; our operations *UNFOLD*, *FOLD*, *GENERALIZE* correspond closely to the *direct tree transformations* or the *m-tree transformations* in the Martens-Gallagher framework.

There are two main differences between their framework and ours. First, in *their* framework one must in every step compare the new atom to the *nearest* ancestor in the same class and make sure that the present atom *is smaller* according to the wfo on that class. If this fails, one must fold or generalize (to use our terminology). In *our* framework one must compare the present term to *all* the ancestors in the same class, but the criterion for stopping is that the present term is *not larger* then any of the preceding ones. (We do not operate with an explicit partition into classes but the definition of $\trianglelefteq$ implicitly has an effect similar to separating atoms into disjoint classes according to the name of the predicate in the atom.)

This gives our technique an advantage in some cases. For instance consider a branch in a partial SLD-tree with a node $q([1,2],1)$ and then subsequently $q(1,[1,2])$. If the well-founded order is based on some form of functor measure in some arguments, then we should stop at the second node if we are measuring the second argument, because its size has grown, but not if we are measuring the first argument since its size has decreased. Thus, whether the chosen functor measure allows us to go on involves an element of chance. On the other hand, using the wqo of our framework one would continue in any case because an argument has decreased. This situation occurs for instance in the KMP example with terms (2) and (4).

The second difference is that the technique of Martens and Gallagher is a general *framework*, where one must somehow choose a wfo, while our technique is an *instance* with one specific wqo. The choice of one specific functor measure in their framework seems to involve some heuristics. On the other hand, we believe that the homeomorphic embedding relation $\trianglelefteq$ captures a universal intuition that any infinite sequence of terms over a finite alphabet must involve some element of *repetition*; $s \trianglelefteq t$ means exactly that s and t are the same, except that in t extra subterms have been added.

It would be interesting to try and use the homeomorphic embedding relation in the Martens-Gallagher framework.

7 Related Work

Supercompilation was formulated in the early seventies by Turchin and has been further developed in the eighties; see [23]. From its very inception, supercompilation has been tied to a specific programming language, called Refal. Our partial process trees are closely related to Refal graphs [23] but due to our data structure, the definition of driving and generalization becomes simpler; *e.g.* most general unifiers do not always exist for Refal data structures (associative and possibly nested symbol strings).

The termination technique described in [24] deals with the problem of nested function calls and uses a sophisticated strategy for structuring the stack to detect recurrent call patterns; it appears that our method is at least as good as that method. Moreover, our whistle treats nested function calls and data structures in a *uniform* way. Later versions of the supercompiler use more refined techniques, but have not been described in the literature.

Termination techniques in logic programming and partial deduction have been of major concern as evidenced by a series of publications, *e.g.* [25, 16, 2, 1, 13, 14]. With the exception of [14], which we discussed in the previous section, the ones dealing with partial deduction address *local termination*, and so do not generally ensure termination.

The present work belongs to a line of work which aims at a better understanding of supercompilation and its relation to other program transformations; *e.g.* [6, 7, 8, 10, 18, 19, 15].

Acknowledgements

The authors are indebted to V.F. Turchin for sharing with us his many insights and to B. Martens for helpful discussions. Thanks to M. Alpuente for insightful comments on a previous draft, to the anonymous referees for constructive feedback, and to the TOPPS group at DIKU for providing a stimulating environment. The second author was partly supported by an Erwin-Schrödinger Fellowship of the Austrian Science Foundation (FWF) under grant J0780 and J0964.

References

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

[2] M. Bruynooghe, D. De Schreye, B. Martens. A General Criterion for Avoiding Infinite Unfolding During Partial Deduction. *New Generation Computing.* 11(1): 47-79, 1992.

[3] N. Dershowitz, J.-P. Jouannaud. Rewrite Systems. *J. van Leeuwen (ed.), Handbook of Theoretical Computer Science.* 244-320, Elsevier 1992.

[4] J. Gallagher. Tutorial on Specialisation of Logic Programs. *PEPM'93.* 88-98, ACM Press 1993.

[5] R. Glück, V.F. Turchin. Application of Metasystem Transition to Function Inversion and Transformation. *ISSAC'90.* 286-287, ACM Press 1990.

[6] R. Glück, A. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. *P. Cousot et al. (eds.), Static Analysis.* LNCS 724, 112-123, Springer-Verlag 1993.

[7] R. Glück, J. Jørgensen. Generating Transformers for Deforestation and Driving. *B. Le Charlier (ed.), Static Analysis.* LNCS 864, 432-448, Springer-Vlg 1994.

[8] R. Glück, M.H. Sørensen. Partial Deduction and Driving are Equivalent. *M. Hermenegildo, et al. (eds.), PLILP'94.* LNCS 844, 165-181, Springer-Vlg 1994.

[9] N.D. Jones, C. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International. 1993

[10] N.D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. *N.D. Jones, et al. (eds.), Logic, Language and Computation.* LNCS 792, 206-224, Springer-Verlag 1994.

[11] J. Komorowski. An Introduction to Partial Deduction. *A. Pettorossi (ed.), Meta-Programming in Logic.* LNCS 649, 49-69, Springer-Verlag 1992.

[12] J.W. Lloyd, J.C. Shepherdson. Partial Evaluation in Logic Programming. *J of Logic Programming.* 11(3-4): 217-242, 1991.

[13] B. Martens, D. De Schreye, T. Horváth. Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures. *TCS.* 122: 97-117, 1994.

[14] B. Martens, J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. *L. Sterling (ed.), ICLP'95.* 597-613, MIT Press 1995.

[15] K. Nielsen, M.H. Sørensen. Call-by-Name CPS-Translation as a Binding-Time Improvement. *Static Analysis.* LNCS, Springer-Verlag (to appear) 1995.

[16] L. Plümer. *Termination Proofs for Logic Programs.* LNAI 446, Springer-Vlg 1990.

[17] D.A. Smith. Partial Evaluation of Pattern Matching in Constraint Logic Programming Languages. *PEPM'91.* 62-71, ACM Press 1991.

[18] M.H. Sørensen. *Turchin's Supercompiler Revisited. An Operational Theory of Positive Information Propagation.* Master's Thesis, DIKU report 94/7, 1994.

[19] M.H. Sørensen, R. Glück, N.D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. *D. Sannella (ed.), Programming Languages and Systems.* LNCS 788, 485-500, Springer-Verlag 1994.

[20] V.F. Turchin. Equivalent Transformations of Recursive Functions Defined in Refal. *Teorija Jazykov i Metody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods). (Kiev-Alushta, USSR).* 31-42, 1972 (in Russian).

[21] V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *J.W. de Bakker, J. van Leeuwen (eds.), Automata, Languages and Programming.* LNCS 85, 645-657, Springer-Verlag 1980.

[22] V.F. Turchin, R. Nirenberg, D. Turchin. Experiments with a Supercompiler. *Symposium on Lisp and Functional Programming.* 47-55, ACM Press 1982.

[23] V.F. Turchin. The Concept of a Supercompiler. *TOPLAS.* 8(3): 292-325, 1986.

[24] V.F. Turchin. The Algorithm of Generalization in the Supercompiler. *A.P. Ershov, D. Bjørner, N.D. Jones (eds.), Partial Evaluation and Mixed Computation.* 341-353, North-Holland 1988.

[25] J.D. Ullman, A. Van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *J of the ACM.* 35(2): 345-373, 1988.

[26] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *TCS.* 73: 231-248, 1990.