On Perfect Supercompilation

Jens Peter Secher and Morten Heine Sørensen

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: {jpsecher,rambo}@diku.dk

Abstract. We extend *positive supercompilation* to handle *negative* as well as positive information. This is done by instrumenting the underlying *unfold rules* with a small rewrite system that handles *constraints on terms*, thereby ensuring *perfect information propagation*. We illustrate this by transforming a naïvely specialised string matcher into an optimal one. The presented algorithm is guaranteed to *terminate* by means of *generalisation steps*.

1 Introduction

Turchin's supercompiler [21] is a program transformer for functional programs which performs optimisations beyond *partial evaluation* [9] and *deforestation* [23].

Positive supercompilation [7] is a variant of Turchin's supercompiler which was introduced in an attempt to study and explain the essentials of Turchin's supercompiler, how it achieves its effects, and its relation to other transformers. In particular, the language of the programs to be transformed by positive supercompilation is a typical first-order functional language — the one usually studied in deforestation — which is rather different from the language Refal, usually adopted in connection with Turchin's supercompiler.

For the sake of simplicity, the positive supercompiler was designed to maintain *positive information* only; that is, when the transformer reaches a conditional **if** $x{=}{=}x'$ **then** t **else** t', the information that $x = x'$ is assumed to hold is taken into account when transforming t (by performing the substitution $\{x := x'\}$ on t). In contrast, the *negative information* that $x \neq x'$ must hold is discarded when transforming t' (since no substitution can represent this information!). In Turchin's supercompiler this negative information is maintained as a constraint when transforming t'. Consequently, Turchin's supercompiler can perform some optimisations beyond positive supercompilation.

In this paper we present an algorithm which we call *perfect supercompilation* — a term essentially adopted from [6] — which is similar to Turchin's supercompiler. The perfect supercompiler arises by extending the positive supercompiler to take negative information into account. Thus, we retain the typical first-order language as the language of programs to be transformed, and we adopt the style of presentation from positive supercompilation.

A main contribution of the extension is to develop techniques which manipulate constraints of a rather general form. Although running implementations of Turchin's supercompiler use such techniques to some extent, the techniques have not been presented in the literature for Turchin's supercompiler as far as we know. The only exception is the paper by Glück and Klimov [6] which, however, handles constraints of a simpler form; for instance, our algorithm for *normalising* constraints has no counterpart in their technique. As another main contribution we generalise a technique for ensuring that positive supercompilation always terminates to the perfect supercompiler and prove that, indeed, perfect supercompilation terminates on all programs[1]. As far as we are aware, no version of Turchin's supercompiler maintaining negative information has been presented which in general is guaranteed to terminate.

The remainder of this paper is organised as follows. We first (Sect. 2) present a classical application of positive supercompilation (of transformers in general): the generation of an efficient specialised string pattern matcher from a general matcher and a known pattern. As is well-known, positive supercompilation generates specialised matchers containing redundant tests. We also show how these redundant tests are eliminated when one uses instead perfect supercompilation. We then (Sect. 3) present an overview of perfect supercompilation and (Sect. 4) an overview of the proof that perfect supercompilation always terminates. In Sect. 5 we conclude and compare to related work.

2 The Knuth-Morris-Pratt Example

In this paper we will only consider programs written in a first-order, functional language with pattern matching and conditionals. For simplicity, pattern-matching functions are allowed to match with non-nested patterns on one parameter only, and conditionals can only be used to test the equality of two values by means of the $==$ operator. We will use the convention that function names are written *slanted*, variables are written in *italics*, and constructors are written in SMALL CAPS. We also use standard shorthand notation $[]$ and $h:t$ for the empty list and the list constructed from h and the tail t, respectively; we further use the usual notation $[h_1, \ldots, h_n]$.

Consider the following *general matcher* program which takes a pattern and a string as input and returns TRUE iff the pattern occurs as a substring in the string.

$$
\begin{array}{ll}
\mathrm{match}(p, s) & = \mathrm{m}(p, s, p, s) \\
\mathrm{m}([\,], ss, op, os) & = \textsc{true} \\
\mathrm{m}(p : pp, ss, op, os) & = \mathrm{x}(p, pp, ss, op, os) \\
\mathrm{x}(p, pp, [\,], op, os) & = \textsc{false} \\
\mathrm{x}(p, pp, s : ss, op, os) & = \mathbf{if}\ p{=}{=}s\ \ \mathbf{then}\ \ \mathrm{m}(pp, ss, op, os)\ \ \mathbf{else}\ \ \mathrm{n}(op, os) \\
\mathrm{n}(op, s : ss) & = \mathrm{m}(op, ss, op, ss)\ \ .
\end{array}
$$

[1] When termination is guaranteed we cannot guarantee perfect transformation, but the underlying unfolding scheme is still perfect in the sense that all information is propagated.

Although this example only compares variables to variables, our method can manipulate more general equalities and inequalities.

Now consider the following *naïvely specialised matcher* $match_{\text{AAB}}$ which matches the fixed pattern $[\text{A}, \text{A}, \text{B}]$ with a string u by calling $match$:

$$match_{\text{AAB}}(u) \;=\; match([\text{A}, \text{A}, \text{B}], u) \;\; .$$

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

This strategy is not optimal. Suppose that after matching the two occurrences of A in the pattern with the first two occurrences of A in the string, the B in the pattern fails to match yet another A in the string. Then the process is restarted with the string's tail, even though it is known that the first two comparisons will succeed. Rather than performing these tests whose outcome is known, we should *skip* the first three occurrences of A in the original string and proceed directly to compare the B in the pattern with the fourth element of the original string. This is done in the *KMP specialised matcher:*

$$
\begin{array}{ll}
match_{\text{AAB}}(u) & = m_{\text{AAB}}(u) \\
m_{\text{AAB}}([\,]) & = \text{FALSE} \\
m_{\text{AAB}}(s:ss) & = \textbf{if } \text{A}{=}{=}s \textbf{ then } m_{\text{AB}}(ss) \textbf{ else } m_{\text{AAB}}(ss) \\
m_{\text{AB}}([\,]) & = \text{FALSE} \\
m_{\text{AB}}(s:ss) & = \textbf{if } \text{A}{=}{=}s \textbf{ then } m_{\text{B}}(ss) \textbf{ else } m_{\text{AAB}}(ss) \\
m_{\text{B}}([\,]) & = \text{FALSE} \\
m_{\text{B}}(s:ss) & = \textbf{if } \text{B}{=}{=}s \textbf{ then } \text{TRUE} \\
 & \qquad\qquad\quad \textbf{else if } \text{A}{=}{=}s \textbf{ then } m_{\text{B}}(ss) \textbf{ else } m_{\text{AAB}}(ss) \;\; .
\end{array}
$$

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

Can we get this program by application of *positive* supercompilation to the naïvely specialised matcher? The result of this application is depicted graphically as a *process tree* in Fig. 1 (it will be explained later why some nodes are emphasised). The root of the process tree is labelled by the initial term that is to be transformed (here, the naïvely specialised matcher). The children of a node α in the process tree represent possible unfoldings for the term in α; the edges are labelled with the assumptions made about free variables (e.g. $u = [\,]$). Each arc in the process tree can therefore be seen as one step of transformation. At the same time the whole tree can be viewed as a new program, where arcs with labels represent tests on the input, and the leaves represent final results or recursive calls.

Informally, a program can be extracted from a process tree by creating a new function definition for each node α that has labelled outgoing edges; the new function has as parameters the set of variables in α and a right-hand side

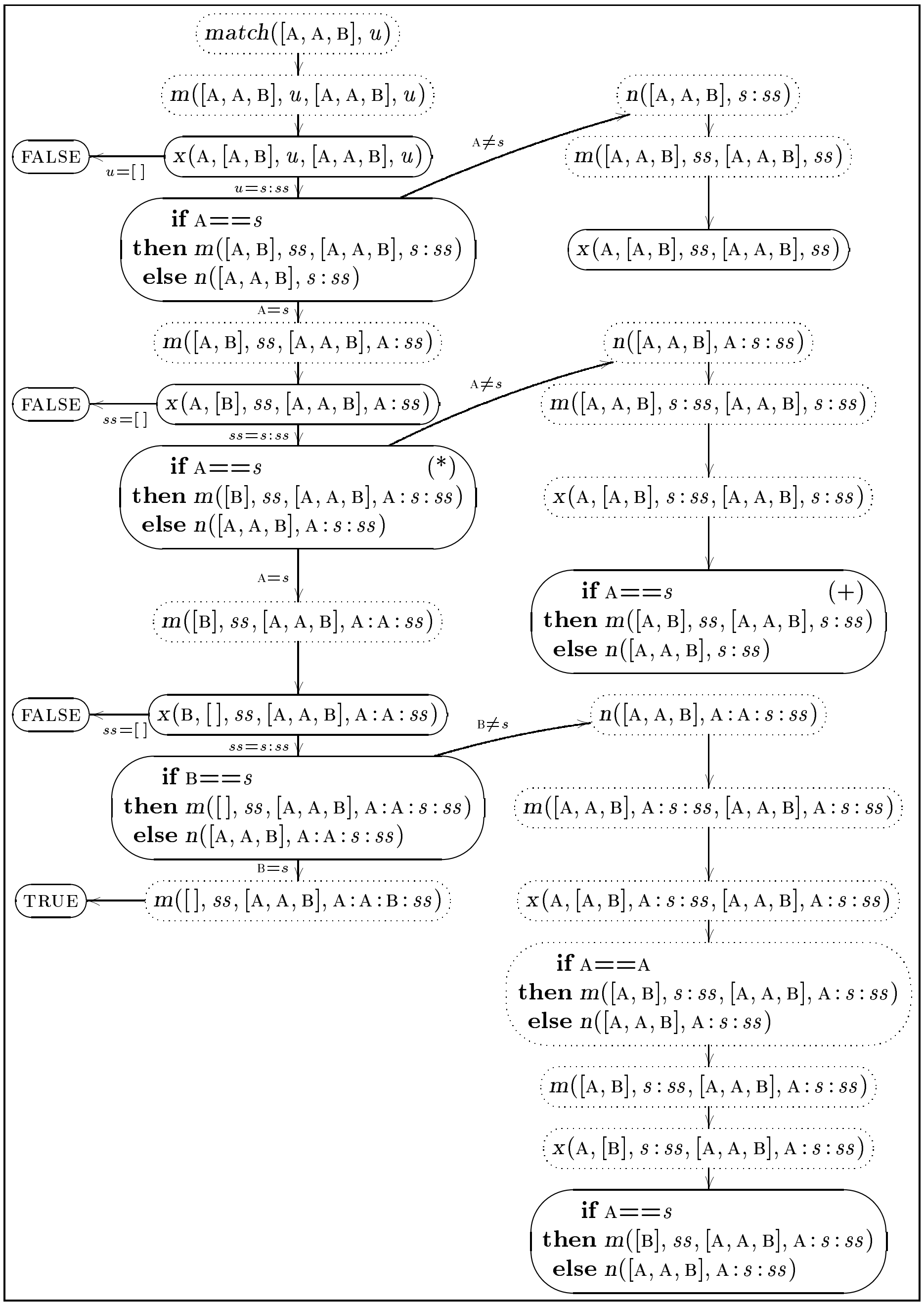


Fig. 1. Driving the naïvely specialised matcher. The children of a node α represent possible unfoldings for the term in α; the edges are labelled with the assumptions made.

created from the children of α. In fact, the program corresponding to the tree in Fig. 1 is the following:

$$
\begin{array}{ll}
m_{\text{AAB}}([\,]) & = \text{FALSE} \\
m_{\text{AAB}}(s:ss) & = \textbf{if } \text{A}{=}{=}s \textbf{ then } m_{\text{AB}}(ss) \textbf{ else } n_{\text{AAB}}(ss, s) \\
m_{\text{AB}}([\,]) & = \text{FALSE} \\
m_{\text{AB}}(s:ss) & = \textbf{if } \text{A}{=}{=}s \textbf{ then } m_{\text{B}}(ss) \textbf{ else } n_{\text{AB}}(ss, s) \\
m_{\text{B}}([\,]) & = \text{FALSE} \\
m_{\text{B}}(s:ss) & = \textbf{if } \text{B}{=}{=}s \textbf{ then } \text{TRUE} \textbf{ else } n_{\text{B}}(ss, s) \\
n_{\text{AAB}}(ss, s) & = m_{\text{AAB}}(ss) \\
n_{\text{AB}}(ss, s) & = \textbf{if } \text{A}{=}{=}s \textbf{ then } m_{\text{AB}}(ss) \textbf{ else } n_{\text{AAB}}(ss, s) \\
n_{\text{B}}(ss, s) & = \textbf{if } \text{A}{=}{=}s \textbf{ then } m_{\text{B}}(ss) \textbf{ else } n_{\text{AB}}(ss, s) \ .
\end{array}
$$

The term $m_{\text{AAB}}(u)$ in this program is more efficient than $match([\text{A}, \text{A}, \text{B}], u)$ in the original program. In fact, this is the desired KMP specialised matcher, except for the redundant test $\text{A}{=}{=}s$ in n_{AB} (and the redundant argument in n_{AAB}). The reason for the redundant test $\text{A}{=}{=}s$ is that positive supercompilation ignores negative information: when proceeding to the false branch of the conditional (from the original program)

$$\textbf{if } \text{A}{=}{=}s \textbf{ then } m([\text{B}], ss, [\text{A}, \text{A}, \text{B}], \text{A}:s:ss) \textbf{ else } n([\text{A}, \text{A}, \text{B}], \text{A}:s:ss) \ , \qquad (*)$$

the information that $\text{A} \neq s$ holds is forgotten. Therefore, the test is repeated in the subsequent conditional

$$\textbf{if } \text{A}{=}{=}s \textbf{ then } m([\text{A}, \text{B}], ss, [\text{A}, \text{A}, \text{B}], s:ss) \textbf{ else } n([\text{A}, \text{A}, \text{B}], s:ss) \ . \qquad (+)$$

In contrast, with perfect supercompilation, this information is maintained as a constraint, and can be used to decide that the conditional (+) has only one possible outcome. The tree would therefore continue below the node (+), and the resulting program would skip the superfluous test and have a recursive call back to the first branching node, $\text{X}(\text{A}, [\text{A}, \text{B}], u, [\text{A}, \text{A}, \text{B}], u)$; this is exactly the KMP specialised matcher.

3 Overview of Perfect Supercompilation

In this and the next section we will give an informal account of the supercompilation algorithm. For proofs, examples and in-depth treatment of the algorithm, see [15].

Despite the intended informality we will need the following definitions. Let x, y, z range over variables from the set X. Let c, f and g range over fixed arity constructor, function and pattern-matching-function names in the finite sets C, F and G, respectively. Let p range over patterns of the form $c(x_1, \ldots, x_n)$ and let t, u, s range over terms. By $\text{var}(t)$ we denote the variables in t, and we let θ range over substitutions, written $\{x_1 := t_1, \ldots, x_n := t_n\}$; application of substitutions is defined as usual and written prefix. Finally, we write $f(\ldots) \triangleq t$ to denote that function f is defined in the program under consideration.

Perfect supercompilation of a program is carried out in two phases. First, a model of the subject program is constructed in form of a *constrained process tree*. Second, a new program is extracted from the constrained process tree. A constrained process tree is similar to the tree in Fig. 1, but each node is labelled by a term *and* a set of *constraints*. From here on we will not distinguish between constrained and unconstrained process trees, and we will refer to some part of the label of a node α simply by saying "node α contains ... ".

The root of the process tree is labelled by the initial term that is to be transformed, together with an empty *constraint system*. The process tree is developed by repeated unfoldings of the terms in the leaves. The rules that govern the unfolding of terms have been constructed by extending the small-step semantics of the language by rules that speculatively execute tests that depend on variables. For each possible outcome of a test, a child is added and information about the test that has been conducted is appended to the current constraint system. The extended constraint system is then passed on to the child that resulted from the speculative execution. Our constraint systems (a subset of the ones defined in [2]) are restricted kinds of conjunctive normal forms of formulae of the form

$$\left(\bigwedge_{i=1}^{n} a_i = a_i'\right) \wedge \left(\bigwedge_{i=1}^{m} b_i \neq b_i'\right)$$

where a, b are terms that consist of variables and constructors only, *i.e.*

$$a, b ::= x \mid c(a_1, \ldots, a_n) \ .$$

The constraint systems are used to prune branches from the process tree: speculative execution of a test that results in a constraint system that cannot be *satisfied* will not produce a new child. For instance, consider the again the conditional (+); blindly unfolding this term would result in a node with two children:

$$\textbf{if } \mathrm{A}{=}{=}s \textbf{ then } m([\mathrm{A},\mathrm{B}], ss, [\mathrm{A},\mathrm{A},\mathrm{B}], s\!:\!ss) \textbf{ else } n([\mathrm{A},\mathrm{A},\mathrm{B}], s\!:\!ss)$$

$$\xrightarrow{\mathrm{A}=s} m([\mathrm{A},\mathrm{B}], ss, [\mathrm{A},\mathrm{A},\mathrm{B}], s\!:\!ss) \qquad\qquad \xrightarrow{\mathrm{A}\neq s} n([\mathrm{A},\mathrm{A},\mathrm{B}], s\!:\!ss)$$

But since we have inherited the constraint system $\mathrm{A} \neq s$ from the conditional (*), the left child will not be produced because the resulting constraint system $\mathrm{A} \neq s \wedge \mathrm{A} = s$ is not satisfiable. More precisely, for a constraint system to be satisfiable, it must be possible to assign values to the variables in the system such that the constraints are satisfied. A constraint system is thus satisfiable iff there exists a substitution θ such that, for each equation $a = a'$, θa will be syntactically identical to $\theta a'$, and likewise, for each disequation $b \neq b'$, θb will be syntactically different from $\theta b'$. To decide the satisfiability of a constraint system R, we first apply a set of rewrite rules to bring R into a *normal form*. The core of these rewrite rules (a modified version of the ones presented in [2]) is shown in Fig. 2. Additional control on these rules ensure that non-deterministic,

$$
\begin{array}{rcll}
x = x & \hookrightarrow & \top & \\
x \neq x & \hookrightarrow & \bot & \\
\bullet \wedge \top & \hookrightarrow & \bullet & \\
\bullet \vee \bot & \hookrightarrow & \bullet & \\
\bullet \wedge \bot & \hookrightarrow & \bot & \\
\bullet \vee \top & \hookrightarrow & \top & \\
c(b_1, \ldots, b_n) = c'(a_1, \ldots, a_m) & \hookrightarrow & \bot & (c \neq c') \\
c(b_1, \ldots, b_n) \neq c'(a_1, \ldots, a_m) & \hookrightarrow & \top & (c \neq c') \\
c(b_1, \ldots, b_n) = c(a_1, \ldots, a_n) & \hookrightarrow & b_1 = a_1 \wedge \cdots \wedge b_n = a_n & \\
c(b_1, \ldots, b_n) \neq c(a_1, \ldots, a_n) & \hookrightarrow & b_1 \neq a_1 \vee \cdots \vee b_n \neq a_n & \\
x = a & \hookrightarrow & \bot & (x \in \mathrm{var}(a) \,\&\, a \notin X) \\
x \neq a & \hookrightarrow & \top & (x \in \mathrm{var}(a) \,\&\, a \notin X) \\
x = a \wedge \bullet & \hookrightarrow & x = a \wedge \bullet\{x := a\} & (x \notin \mathrm{var}(a)) \\
x \neq a \vee \bullet & \hookrightarrow & x \neq a \vee \bullet\{x := a\} & (x \notin \mathrm{var}(a))
\end{array}
$$

Fig. 2. Rewrite system for normalisation of constraint systems. $\bot$ represents an unsatisfiable element, $\top$ represents the trivially satisfiable element, and $\bullet$ stands for an arbitrary part of a formula.

exhaustive application of the rewrite rules to any constraint system terminates and results in a constraint system in normal form. A constraint system in normal form is either $\bot$ (false), $\top$ (true), or of the form

$$\left(\bigwedge_{i=1}^{n} x_i = a_i \right) \wedge \bigwedge_{j=1}^{m} \left(\bigvee_{k=1}^{l} y_{j,k} \neq b_{j,k} \right) \quad .$$

When no *type information* about the variables in a constraint system is present, a constraint system in normal form is satisfiable exactly when it is different from $\bot$. However, when it is known that a variable x can assume a *finite* set of values only, it is necessary to verify that there indeed exists a *value* which, when assigned to x, satisfies the constraint system. For instance, consider the constraint system

$$x \neq y \wedge y \neq z \wedge z \neq x$$

where all variables have boolean type. This system is in normal form and therefore appears to be satisfiable — but it is not possible to assign values FALSE or TRUE to the variables such that the system is satisfiable. When a constraint system R is in normal form, it is thus necessary to systematically try out all possible combinations of value assignments for variables with known, finitely-valued types. This is done by instantiating such variables in R, which possibly will call for further rewrite steps to take R into a normal form, and so on until there are no more finitely-valued variables left. If R now is different from $\bot$, R is satisfiable [15].

We can now show how constraint systems can be used to guide the construction of the process tree. Every term t in the process tree is associated with a constraint system R, denoted $\langle t, R\rangle$. The complete set of unfold rules is presented in Fig. 3. Rules (A)–(B), (D), (E)–(F)[2] and (G)–(J) correspond to normal evaluation with respect to the semantics of language[3]. Rules (C) and (E)–(F)[4] perform speculative execution of a term based on the information in the associated constraint system.

Rule (C) instantiates a free variable y to the pattern $c(y_1, \ldots, y_m)$ taken from the function definition (using fresh variables). This is achieved by appending the equation $y = c(y_1, \ldots, y_m)$ to the current constraint system. If the new constraint system is satisfiable, the function application can be unfolded. In the same manner, rules (E) and (F) handle conditional expressions where general equations and disequations can be appended to the constraint system. Rule (K) finally separates the resulting constraint system R into positive and negative information by normalising R: the positive information (of the form $\bigwedge x = a$) can be regarded as a substitution, which can then be separated from the normalised R; we denote this separation by $R' \cong (\theta \wedge R'')$. The positive information can then be propagated to the context by applying the substitution θ to the whole term[5].

Unfolding of a branch is stopped if the leaf in that branch is a value or if an ancestor node *covers* (explained below) all possible executions that can arise from the leaf. The latter case constitutes a *fold* operation which will eventually result in a recursive call in the derived program.

We say that a node covers another node if the terms of the two nodes are equal up to renaming of variables *and* the constraint system in the leaf is at least as restrictive as the one in its ancestor. Intuitively speaking, if these two conditions are met, any real computation performed by the leaf can also be performed by the ancestor; we can then safely produce a recursive call in the derived program. In [15] an algorithm is presented that gives a safe approximation to the question "is R more restrictive than R'?".

If we look at the process tree in Fig. 1, we will see that some parts of the tree are created by *deterministic unfolding*, *i.e.* they each consist of a single path. This is a good sign, since it means that the path represents local computations that will *always* be carried out when the program is in this particular state, regardless of the uninstantiated variables. We have thus precomputed these intermediate transitions once and for all — as done in partial evaluation — and we can omit the intermediate steps and simply remember the result.

Creation of a process tree in the manner just described does not always terminate since infinite process trees can be produced. To keep the process trees finite, we ensure that no infinite branches are produced. It turns out that in

[2] Without free variables in a and a'.

[3] The intended semantics of our language is evaluation to weak head normal form, except for comparison in conditionals where the terms to be compared are *fully* evaluated before the comparison is carried out. For simplicity, the unfolding rules are call-by-name which, unfortunately, can give rise to duplication of computation.

[4] With free variables in a or a'.

[5] The positive information is re-injected into the context because such re-injection neatly disposes of irrelevant information about variables that are no longer present.

$$\boxed{\langle t, R\rangle \mapsto \langle t', R'\rangle}$$

$$\text{(A)}\ \frac{f(x_1,\ldots,x_n) \triangleq t}{\langle f(t_1,\ldots,t_n), R\rangle \mapsto \langle \{x_1 := t_1,\ldots,x_n := t_n\}t, R\rangle}$$

$$\text{(B)}\ \frac{g(c(x_1,\ldots,x_m), x_{m+1},\ldots,x_n) \triangleq t}{\langle g(c(t_1,\ldots,t_m), t_{m+1},\ldots,t_n), R\rangle \mapsto \langle \{x_1 := t_1,\ldots,x_n := t_n\}t, R\rangle}$$

$$\text{(C)}\ \frac{g(p, x_1,\ldots,x_n) \triangleq t \qquad R' = R \wedge [x = p] \qquad \text{satisfiable}(R')}{\langle g(x, t_1,\ldots,t_n), R\rangle \mapsto \langle \{x_1 := t_1,\ldots,x_n := t_n\}t, R'\rangle}$$

$$\text{(D)}\ \frac{g(p, x_1,\ldots,x_n) \triangleq t \qquad \langle t, R\rangle \mapsto \langle t', R'\rangle}{\langle g(t, t_1,\ldots,t_n), R\rangle \mapsto \langle g(t', t_1,\ldots,t_n), R'\rangle}$$

$$\text{(E)}\ \frac{R' = R \wedge [a = a'] \qquad \text{satisfiable}(R')}{\langle \textbf{if } a == a' \textbf{ then } t \textbf{ else } t', R\rangle \mapsto \langle t, R'\rangle}$$

$$\text{(F)}\ \frac{R' = R \wedge [a \neq a'] \qquad \text{satisfiable}(R')}{\langle \textbf{if } a == a' \textbf{ then } t \textbf{ else } t', R\rangle \mapsto \langle t', R'\rangle}$$

$$\text{(G)}\ \frac{\langle t_1, R\rangle \Mapsto \langle t_1', R'\rangle}{\langle \textbf{if } t_1 == t_2 \textbf{ then } t_3 \textbf{ else } t_4, R\rangle \mapsto \langle \textbf{if } t_1' == t_2 \textbf{ then } t_3 \textbf{ else } t_4, R'\rangle}$$

$$\text{(H)}\ \frac{\langle t_2, R\rangle \Mapsto \langle t_2', R'\rangle}{\langle \textbf{if } a == t_2 \textbf{ then } t_3 \textbf{ else } t_4, R\rangle \mapsto \langle \textbf{if } a == t_2' \textbf{ then } t_3 \textbf{ else } t_4, R'\rangle}$$

$$\boxed{\langle t, R\rangle \Mapsto \langle t', R'\rangle}$$

$$\text{(I)}\ \frac{\langle t, R\rangle \mapsto \langle t', R'\rangle}{\langle t, R\rangle \Mapsto \langle t', R'\rangle}$$

$$\text{(J)}\ \frac{\langle t, R\rangle \Mapsto \langle t', R'\rangle}{\langle c(a_1,\ldots,a_{m-1}, t, t_{m+1},\ldots,t_n), R\rangle \Mapsto \langle c(a_1,\ldots,a_{m-1}, t', t_{m+1},\ldots,t_n), R'\rangle}$$

$$\boxed{\langle t, R\rangle \Rrightarrow \langle t', R'\rangle}$$

$$\text{(K)}\ \frac{\langle t, R\rangle \Mapsto \langle t', R'\rangle \qquad R' \cong (\theta \wedge R'')}{\langle t, R\rangle \Rrightarrow \langle \theta t', R''\rangle}$$

Fig. 3. Unfold rules with perfect information propagation.

every infinite branch, there must be a term that *homeomorphically embeds* an ancestor (this is known as Kruskal's Tree Theorem). The homeomorphic embedding relation $\trianglelefteq$ is the smallest relation on terms such that, for any symbol $h \in C \cup F \cup G \cup \{\textbf{ifthenelse}\}$,

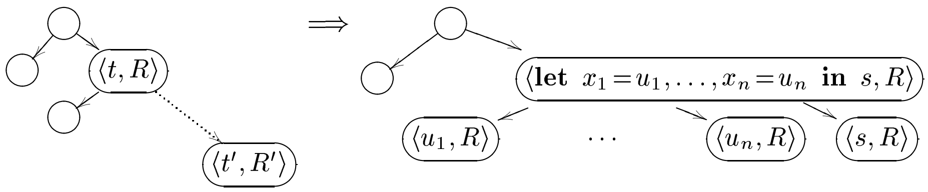
$$\frac{}{x \trianglelefteq y} \qquad \frac{\exists i \in \{1,\ldots,n\} : t \trianglelefteq t'_i}{t \trianglelefteq h(t'_1,\ldots,t'_n)} \qquad \frac{\forall i \in \{1,\ldots,n\} : t_i \trianglelefteq t'_i}{h(t_1,\ldots,t_n) \trianglelefteq h(t'_1,\ldots,t'_n)} \; .$$

When a term t' in a leaf homeomorphically embeds a term t in an ancestor, there is thus a danger of producing an infinite branch. In such a situation, t or t' is split up by means of a *generalisation step*.

Definition 1 (Generalisation).

1. A term u is an *instance* of term t, denoted $u \geq t$, if there exists a substitution θ such that $\theta t = u$.
2. A generalisation of two terms t, u is a term s such that $t \geq s$ and $u \geq s$.
3. A *most specific generalisation* (msg) of two terms t, u is a generalisation s such that, for all generalisation s' of t, u, $s \geq s'$. (There exists exactly one msg of t, u modulo renaming). □

A generalisation step on a process tree calculates the msg of the terms t, t' in two nodes α, α'; the msg is then used to divide one of the nodes into subterms that can be unfolded independently:



where s is the msg of t and t', and $t = s\{x_1 := u_1, \ldots, x_n := u_n\}$. Which of the nodes t, t' that is split up depends on how similar the nodes are (this will be made more precise below). The introduction of **let**-terms are merely for notational convenience; they will be unrolled in the derived program.

We can now sketch the full supercompilation algorithm. To ensure termination and, at the same time, provide reasonable specialisation, we partition the nodes in the process tree into three categories (in the lines of [18]):

1. nodes containing **let**-terms,
2. *global* nodes, and
3. *local* nodes.

Global nodes are those that represent speculative execution or final results (both of which must be present in the derived program). Local nodes are those nodes that are not global and does not contain **let**-terms. For example, in Fig. 1 the set of local nodes are indicated by dotted frames (there are no nodes containing **let**-terms since there is no need for generalisation in that particular example). This partitioning of the nodes is used to control the unfolding.

Definition 2 (Relevant ancestor). Let T be a process tree and let the set of *relevant ancestors* $\text{relanc}(T, \alpha)$ of a node α in T be defined thus:

$$\text{relanc}(T, \alpha) = \begin{cases} \emptyset, & \text{if } \alpha \text{ contains a } \textbf{let}\text{-term} \\ \text{all ancestors that are global}, & \text{if } \alpha \text{ is global} \\ \text{all local ancestors}, & \text{if } \alpha \text{ is local} \end{cases}$$

where the *local ancestors* to α is all ancestors that are local up to the first common ancestor that is global. □

For an example, consider the process tree in Fig. 1; the local node $x(\text{A}, [\text{B}], s : ss, [\text{A}, \text{A}, \text{B}], \text{A} : s : ss)$ near the bottom has as local ancestors all ancestors up to and including the node $n([\text{A}, \text{A}, \text{B}], \text{A} : \text{A} : s : ss)$.

Definition 3 (Drive). Let T be a process tree and α a node in T. Then

1. $T(\alpha)$ denotes the label of node α.
2. $T\{\alpha := T'\}$ denotes a new tree that is identical to T except that the subtree rooted at α has been replaced by T'.
3. ϵ denotes the root node of a tree.
4. If $\{\langle t_1, R_1\rangle, \ldots, \langle t_n, R_n\rangle\} = \{\langle t, R\rangle \mid T(\alpha) \Rrightarrow \langle t, R\rangle\}$, then

$$\text{drive}(T, \alpha) = T\{\alpha := \begin{array}{c} T(\alpha) \\ \swarrow \quad \downarrow \quad \searrow \\ \langle t_1, R_1\rangle \quad \cdots \quad \langle t_n, R_n\rangle \end{array}\}$$

□

Definition 4 (Finished). A leaf α in a process tree T is *finished* if one of the following conditions are satisfied:

1. $T(\alpha) = \langle c(), \ldots\rangle$ for some constructor c.
2. $T(\alpha) = \langle x, \ldots\rangle$ for some variable x.
3. There is an ancestor α' to α such that (a) α, α' are global nodes, and (b) if $T(\alpha) = \langle t, R\rangle$ and $T(\alpha') = \langle t', R'\rangle$ then t is a renaming of t' and R' is at least as restrictive as R.

A tree T is said to be *finished* when all leaves are finished. □

With these definitions, we can sketch the supercompilation algorithm thus:

> **input** a term t
> **let** T consist of a single node labelled $\langle t, \top\rangle$
> **while** T is not finished **begin**
>   **let** $\alpha = \langle t, R\rangle$ be an unfinished leaf in T
>   **if** $\forall \alpha' = \langle t', R'\rangle \in \text{relanc}(T, \alpha) : t' \ntrianglelefteq t$ **then** $T = \text{drive}(T, \alpha)$
>   **else begin**
>   **let** $\alpha' = \langle t', R'\rangle \in \text{relanc}(T, \alpha)$ such that $t' \trianglelefteq t$
>   **if** R' is more restrictive than R **then** $T = T\{\alpha' := \langle t', \top\rangle\}$
>   **else if** $t \gtrsim t'$ **then** $T = T\{\alpha := \langle \text{generalise}(t, t'), R\rangle\}$
>   **else** $T = T\{\alpha' := \langle \text{generalise}(t', t), R'\rangle\}$
>   **end**
> **end**
> **output** T

The transformed program can be extracted from the process tree by examination of the global nodes (collecting the set of free variables) and the labels on the edges.

4 Overview of the Termination Proof

A language-independent framework for proving termination of *abstract program transformers* has been presented in [17], where sufficient conditions have been established for abstract program transformers to terminate. In this section we will give a very rough sketch of how this framework have been used to prove termination of our algorithm.

An abstract program transformer is a map from trees to trees, such that a single step of transformation is carried out by each application of the transformer. Termination then amounts to a certain form of convergence of the sequences of trees obtained by repeatedly applying the transformer.

For a transformer to fit the framework, it is sufficient to ensure that

1. the transformer converges, in the sense that for each transformation step, the difference between two consecutive trees lessens. More precisely, in the sequence of trees produced by the transformation, for any depth d there must be some point from which every two consecutive trees are identical down to depth d; and
2. the transformer maintains some invariant such that only finite trees are produced.

By induction on the depth of the trees produced, the former can be proved by the fact that the algorithm either

1. adds new leaves to a tree which trivially makes consecutive trees identical at an increasing depth, or
2. generalises a node, *i.e.* replaces a subtree by node containing a let-term (or a node containing the empty constraint system $\top$). Since generalisation only occurs on terms which are not let-terms (or contain non-empty constraint systems, respectively), a node can be generalised at most twice.

The latter is ensured because, in every proces tree,

1. a path that consists of let-terms only, must be finite since each let-term t will have subsets of t as children proper; thus the size of the nodes in such a path strictly decreases.
2. all other nodes are not allowed to homeomorphically embed an ancestors (except for finished nodes, but these are all leaves).

5 Conclusion and Related Work

We have presented an algorithm for a supercompiler for a first-order functional language that maintains positive as well as negative information. The algorithm is guaranteed to terminate on all programs, it is strong enough to pass the so-called KMP-test.

In [22], Turchin briefly describes how the latest version of his supercompiler utilises *contraction* and *restriction* patterns in driving Refal graphs, the underlying representation of Refal programs. It seems that the resolution of clashes between assignments and contractions/restrictions can achieve propagation of negative information that — to some extent — provides the power equivalent to what has been presented in the present paper, but the exact relationship is at present unclear to us.

In the field of partial evaluation, Consel and Danvy [3] have described how negative information can be incorporated into a naïvely specialised matcher, thereby achieving effects similar to those described in the present paper. This, however, is achieved by a non-trivial rewrite of the subject program before partial evaluation is applied, thus rendering full automation difficult.

In the case of Generalised Partial Computation [5], Takano has presented a transformation technique [19] that exceeds the power of both Turchin's supercompiler and perfect supercompilation. This extra power, however, stems from an unspecified theorem prover that needs to be fed the properties about primitive functions in the language, axioms for the data structures employed in the program under consideration, etc. In [20] the theorem prover is replaced by a congruence closure algorithm [14], which allows for the automatic generation of a KMP-matcher from a naïvely specialised algorithm when some properties about list structures are provided. In comparison to supercompilation, Generalised Partial Computation as formulated by Takano has no concept of generalisation and will therefore terminate only for a small class of programs.

When one abandons simple functional languages (as treated in the present paper) and considers logic programming and constraint logic programming, several accounts exist of equivalent transformation power, *e.g.* [16, 8, 10, 11]. In these frameworks, search and/or constraint solving facilities of the logic language provides the necessary machinery to avoid redundant computations. In this field, great efforts have been made to produce optimal specialisation, and at the same time to ensure termination, see *e.g.* [12, 13].

Acknowledgements. Thanks to Robert Glück, Neil D. Jones, Laura Lafave and Michael Leuschel for discussions and comments. Thanks to Peter Sestoft for many insightful comments to [15].

References

1. ACM. *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26(9) of *ACM SIGPLAN Notices*, New York, September 1991. ACM Press.
2. Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3–4):371–425, March–April 1989.
3. Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

4. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
5. Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151, Amsterdam, 1988. North-Holland.
6. R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
7. R. Glück and M.H. Sørensen. A roadmap to metacomputation by supercompilation. In Danvy et al. [4], pages 137–160.
8. T.J. Hickey and D. Smith. Toward the partial evaluation of CLP languages. In *PEPM'91* [1], pages 43–51.
9. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
10. L. Lafave and J. P. Gallagher. Partial evaluation of functional logic programs in rewriting-based languages. Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, March 1997.
11. L. Lafave and J. P. Gallagher. Extending the power of automatic constraint-based partial evaluators. *ACM Computing Surveys*, 30(3es), September 1998. Article 15.
12. Michael Leuschel and Danny De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*, 1997.
13. Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
14. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.
15. J. P. Secher. Perfect supercompilation. Technical Report 99/01, Department of Computer Science, University of Copenhagen, 1999.
16. D. Smith. Partial evaluation of pattern matching in constraint logic programming. In *PEPM'91* [1], pages 62–71.
17. M.H.B. Sørensen. Convergence of program transformers in the metric space of trees. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 315–337. Springer-Verlag, 1998.
18. M. H. Srensen and R. Glück. Introduction to supercompilation. In *DIKU Summer school on Partial Evaluation*, Lecture Notes in Computer Science. Springer-Verlag, to appear.
19. A. Takano. Generalized partial computation for a lazy functional language. In *PEPM'91* [1], pages 1–11.
20. A. Takano. Generalized partial computation using disunification to solve constraints. In M. Rusinowitch and J.L. Remy, editors, *Conditional Term Rewriting Systems. Proceedings*, volume 656 of *Lecture Notes in Computer Science*, pages 424–428. Springer-Verlag, 1993.
21. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
22. V.F. Turchin. Metacomputation: Metasystem transition plus Supercompilation. In Danvy et al. [4], pages 481–510.
23. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.