A Roadmap to Metacomputation by Supercompilation*

Robert Glück & Morten Heine Sørensen

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

Abstract. This paper gives a gentle introduction to Turchin's supercompilation and its applications in metacomputation with an emphasis on recent developments. First, a complete supercompiler, including positive driving and generalization, is defined for a functional language and illustrated with examples. Then a taxonomy of related transformers is given and compared to the supercompiler. Finally, we put supercompilation into the larger perspective of metacomputation and consider three metacomputation tasks: specialization, composition, and inversion.

Keywords: Program transformation, supercompilation, driving, generalization, metacomputation, metasystem transition.

1 Introduction

Over the years a number of automatic program transformers have been devised and implemented. The most popular is *partial evaluation* which performs *program specialization*. The possibility, in principle, of partial evaluation is contained in Kleene's s–m–n Theorem [41]. The idea to use partial evaluation as a *programming tool* can be traced back to work beginning in the late 1960's by Lombardi and Raphael [50, 49], Dixon [15], and Chang and Lee [10]. Important contributions were made in the seventies by Futamura [19, 20], by Sandewall's group [5], by Ershov [16, 17], and later by Jones' group [39, 40]. In the eighties program specialization became a research field of its own, *e.g.* [7, 46, 38, 12].

Supercompilation [75], conceived by Turchin in the early seventies in Russia for the programming language Refal, achieves the effects of partial evaluation as well as more dramatic optimizations. Turchin formulated the transformations necessary for supercompilation, including the central *rule of driving* and the *outside-in strategy*, in 1972 [68, 69] and the main results concerning self-application, *metasystem transition*, in 1973. The book [83] defined all three *Futamura projections* in terms of metasystem transition. In the English language, the work on supercompilation was first described in [71, 72, 73, 74] and then developed further in [86, 75, 77, 81]. Despite these remarkable contributions, supercompilation has not found recognition outside a small circle of experts.

* Partially supported by the DART project funded by the Danish Natural Sciences Research Council. The first author was also supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foundation (FWF) under grant J0780 & J0964.

This paper gives a gentle introduction to the principles of supercompilation in terms of a *positive supercompiler* [29, 62, 64, 65, 63] comprising two components, *driving* (Sect. 2) and *generalization* (Sect. 3). The supercompiler is compared to related program transformers (Sect. 4), and put it into the larger perspective of metacomputation (Sect. 5). We give references to the literature throughout the text, which can hopefully be used as a starting point for further reading. The bibliography contains a comprehensive list of Russian and English titles on the topic.

1.1 Object Language

We are concerned with a first-order functional language; the intended operational semantics is normal-order graph reduction to weak head normal form in the sense of Bird and Wadler [6].

The syntax of our language appears in Fig. 1 (where $m > 0, n \geq 0$). We assume denumerable, disjoint sets of symbols for variables $v \in \mathcal{V}$, constructors $c \in \mathcal{C}$, and functions $f \in \mathcal{F}$ and $g \in \mathcal{G}$; symbols all have fixed arity. A given program makes use of a finite number of different symbols.

A program $q \in \mathcal{Q}$ is a sequence of function definitions $d \in \mathcal{D}$ where the right side of each definition is a term $t \in \mathcal{T}$ constructed from variables, constructors, function calls, and conditionals. 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 pattern, and that all variables on the right side of a definition be present in its left side. Figure 2 shows the function a for appending two lists.[2]

$\mathcal{Q} \ni q$	$::=$	$d_1 \ldots d_m$	(program)
$\mathcal{D} \ni d$	$::=$	$f\, v_1 \ldots v_n \doteq t$	(f-function definition, no patterns)
	$\mid$	$g\, p_1\, v_1 \ldots v_n \doteq t_1$ $\vdots$ $g\, p_m\, v_1 \ldots v_n \doteq t_m$	(g-function definition with patterns)
$\mathcal{T} \ni t$	$::=$	v	(variable)
	$\mid$	$c\, t_1 \ldots t_n$	(constructor)
	$\mid$	$f\, t_1 \ldots t_n$	(f-function call)
	$\mid$	$g\, t_0\, t_1 \ldots t_n$	(g-function call)
	$\mid$	**if** $t_1{=}t_2$ **then** t_3 **else** t_4	(conditional with equality test)
$\mathcal{P} \ni p$	$::=$	$c\, v_1 \ldots v_n$	(flat pattern)

Fig. 1. Syntax of programs, definitions, terms, and patterns.

Remark. Our language contained case-expressions in [63, 65], g-functions in [64]. The connection of positive supercompilation and deforestation stands out clearest for case-expressions. However, g-functions ("named case-expressions") [18] lead to a simpler presentation of generalization.

[2] We use the usual notation [] and $(x : xs)$ for the list constructors *nil* and *cons x xs*.

$$
\begin{array}{ll}
a\,[\,]\,ys & \doteq ys \\
a\,(x : xs)\,ys & \doteq x : a\,xs\,ys
\end{array}
$$

Fig. 2. Example program *append*.

2 Driving

Driving takes a term and a program and constructs a possibly infinite *process tree*, representing all possible computations with the term, in a certain sense. Figure 3 shows part of the infinite process tree for the term $a\ (a\ xs\ ys)\ xs$ (note the repeated variable xs).

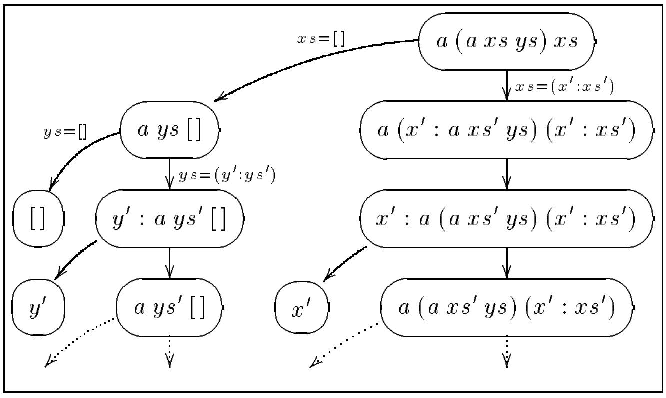


Fig. 3. Example process tree for $a\ (a\ xs\ ys)\ xs$.

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 there are several children so as to account for all possibilities. For instance, the topmost branching in Fig. 3 corresponds to the cases $xs = [\,]$ and $xs = (x' : xs')$.

In Sect. 2.1 we define normal-order reduction, and in Sect. 2.2 we introduce process trees and define driving.

2.1 Normal-Order Reduction

A *value* is a term built exclusively from constructors and variables. An *observable* is either a variable or a term with a known outermost constructor. Any term which is not an observable can be decomposed into 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.

More precisely, define *values*, *observables*, *redexes*, and *evaluation contexts* by the syntactic classes $\mathcal{B}, \mathcal{O}, \mathcal{R}$ and $\mathcal{E}$, respectively, as in Fig. 4. Define $e[t]$ to be the result of substituting t for the "hole" $\diamond$ in e.

Lemma 1 (the unique decomposition property). For any $t \in \mathcal{T}$ either there exists a unique pair $(e, r) \in \mathcal{E} \times \mathcal{R}$ such that $t \equiv e[r]$ or $t \in \mathcal{O}$.

$$
\begin{array}{rcll}
\mathcal{B} \ni b & ::= & v \mid c\, b_1 \ldots b_n & \text{(value)} \\
\mathcal{O} \ni o & ::= & v \mid c\, t_1 \ldots t_n & \text{(observable)} \\[1ex]
\mathcal{R} \ni r & ::= & f\, t_1 \ldots t_n & \text{(redex)} \\
 & \mid & g\, o\, t_1 \ldots t_n & \\
 & \mid & \textbf{if } b_1{=}b_2 \textbf{ then } t_1 \textbf{ else } t_2 & \\[1ex]
\mathcal{E} \ni e & ::= & \diamond & \text{(evaluation context)} \\
 & \mid & g\, e\, t_1 \ldots t_n & \\
 & \mid & \textbf{if } d{=}t_2 \textbf{ then } t_3 \textbf{ else } t_4 & \\
 & \mid & \textbf{if } b{=}d \textbf{ then } t_3 \textbf{ else } t_4 & \\
d & ::= & e \mid c\, b_1 \ldots b_{i-1}\, d\, t_{i+1} \ldots t_n &
\end{array}
$$

Fig. 4. Values, observables, redexes, evaluation contexts.

Figure 5 shows example decompositions. In (1) the outermost call to the f-function f can be unfolded; the evaluation context is empty. In (2) the call to f has to be unfolded; the call to the g-function g cannot be unfolded because the term $f\ t$ does not have a known outermost constructor. In (3) the call to f has to be unfolded; both sides of an equality test must be values.

	t	e	r
(1)	$f\ (g\ t)$	$\diamond$	$f\ (g\ t)$
(2)	$g\ (f\ t)$	$g\,\diamond$	$f\ t$
(3)	$\textbf{if } x{=}c\ (f\ t) \textbf{ then } t' \textbf{ else } t''$	$\textbf{if } x{=}c\,\diamond \textbf{ then } t' \textbf{ else } t''$	$f\ t$

Fig. 5. Examples of decomposition into redex and evaluation context.

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 in Fig. 6. The rules of $\mathcal{N}$ are mutually exclusive and together exhaustive by the unique decomposition property.

t	$\mathcal{N}[\![\, t \,]\!]$
(1) x	$\langle\,\rangle$
(2) $c\ t_1 \ldots t_n$	$\langle t_1, \ldots, t_n \rangle$
(3) $e[f\ t_1 \ldots t_n]$	$\langle e[s\{v_i := t_i\}_{i=1}^n] \rangle$ if $f\ v_1 \ldots v_n \doteq s$
(4) $e[g\ (c\ t_1 \ldots t_i)\ t_{i+1} \ldots t_n]$	$\langle e[s\{v_i := t_i\}_{i=1}^n] \rangle$ if $g\ (c\ v_1 \ldots v_i)\ v_{i+1} \ldots v_n \doteq s$
(5) $e[g\ x\ t_1 \ldots t_n]$	$\langle (e[s_1\{v_i := t_i\}_{i=1}^n])\{x{:=}p_1\}, \ldots, (e[s_m\{v_i := t_i\}_{i=1}^n])\{x{:=}p_m\} \rangle$ if $\{g\ p_j\ v_1 \ldots v_n \doteq s_j\}_{j=1}^m$
(6) $e[\textbf{if } b{=}b' \textbf{ then } t \textbf{ else } t']$	$\begin{cases} \langle e[t] \rangle & \text{if } b, b' \text{ are ground, } b \equiv b' \\ \langle e[t'] \rangle & \text{if } b, b' \text{ are ground, } b \not\equiv b' \\ \langle (e[t])\lceil b, b' \rceil, e[t'] \rangle & \text{if } b, b' \text{ are not both ground} \end{cases}$

Fig. 6. Normal-order reduction step.

Notation: the expression $t\{v_i := t_i\}_{i=1}^n$ denotes the result of simultaneously replacing all occurrences of v_i by the corresponding term t_i where $1 \leq i \leq n$. The expression $\lceil b, b' \rceil$ denotes an idempotent most general unifier $\{v_i := t_i\}_{i=1}^n$ of b and b' if it exists, and *fail* otherwise, where we stipulate $t\ \mathit{fail} \equiv t$. A value is *ground* if it contains no variables. To avoid name capture, the variables occurring in left hand sides in clauses (3)-(5) must be fresh.

Note the substitutions in clauses (5) and (6). The assumed outcome of the test is propagated to the terms resulting from the step. We call this *unification-based* information propagation (*c.f.* Sect. 4).

2.2 Process Trees

A *process tree* is a tree where each node is labeled with a term t and all edges leaving a node are ordered. Every node may have an additional *mark*.

Definition 2. Let T be a process tree and (t) an unmarked leaf node in T. Then $\mathit{UNFOLD}(T, (t))$ is the process tree[3] obtained by marking (t) and adding n unmarked children labeled $t_1, \ldots, t_n$, where $\mathcal{N}[\![t]\!] = \langle t_1, \ldots, t_n \rangle$.

Driving is the action of constructing process trees using two essential principles: normal-order strategy and unification-based information propagation.

Algorithm 3 (driving.)

1. INPUT $t_0 \in \mathcal{T}$, $q \in \mathcal{Q}$
2. LET T_0 be the process tree with unmarked node labeled t_0. SET $i = 0$.
3. WHILE there exists an unmarked leaf node N in T_i:
   (a) $T_{i+1} = \mathit{UNFOLD}(T_i, N)$
   (b) SET $i = i + 1$
4. OUTPUT T_i

3 A Positive Supercompiler

In the previous section we used driving to construct a potentially infinite process tree. The purpose of *generalization* is to ensure that one constructs instead a finite *partial process tree* from which a new term and program can be recovered.

The idea is that if a leaf node M has an ancestor L and it "seems likely" that continued driving will generate an infinite sequence $L, \ldots, M, \ldots$ then M should not be driven any further; instead we should perform *generalization*. In Subsect. 3.1 we define a criterion, a so-called *whistle*, that formalizes the decision when to stop. In Subsect. 3.2 we introduce some notions that are used in Subsect. 3.3 to define generalization. This culminates in a definition of a *positive supercompiler*.

[3] *UNFOLD* and subsequent operations appear in graphical form in Appendix A.

3.1 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 [14]. Variants of this relation are used in termination proofs for term rewrite systems [14] and for ensuring local termination of partial deduction [8]. After it was taken up in [63], it has inspired more recent work [4, 47, 82].

The rationale behind this relation is that in any infinite sequence $t_0, t_1, \ldots$ that arises during driving of a program, there *definitely* exists some $i < j$ with $t_i \trianglelefteq t_j$, so driving cannot proceed infinitely. Moreover, 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, so driving will be stopped for a good reason.

The *homeomorphic embedding* $\trianglelefteq$ is the smallest relation on $\mathcal{T}$ satisfying the rules in Fig. 7, where $h \in \mathcal{X} \cup \mathcal{C} \cup \mathcal{F} \cup \mathcal{G} \cup \{\mathbf{ifthenelse}\}$, $x, y \in \mathcal{X}$, and $s, s_i, t \in \mathcal{T}$.

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

Fig. 7. Homeomorphic embedding.

Diving detects a subterm embedded in a larger term, and *coupling* matches the subterms of two terms. Some examples and non-examples appear in Fig. 8. It is not hard to give an algorithm $\mathit{WHISTLE}(M, N)$ deciding whether $M \trianglelefteq N$.

$b \trianglelefteq a(b)$	$a(c(b)) \not\trianglelefteq c(b)$
$c(b) \trianglelefteq c(a(b))$	$a(c(b)) \not\trianglelefteq c(a(b))$
$d(b, b) \trianglelefteq d(a(b), a(b))$	$a(c(b)) \not\trianglelefteq a(a(a(b)))$

Fig. 8. Examples and non-examples of embedding.

3.2 Most Specific Generalization

We define the generalization of two terms t_1, t_2 as the most specific generalization (msg) $\lfloor t_1, t_2 \rfloor$. A well-known result in term algebra states that any two $t, s \in \mathcal{T}$ have an msg which is unique up to renaming. Examples are shown in Fig. 9.

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

Fig. 9. Examples of most specific generalization.

Definition 4 (instance, generalization, msg, distinct). Given $t_1, t_2 \in \mathcal{T}$.

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 $t_g\theta_1 \equiv t_1$ and $t_g\theta_2 \equiv t_2$.
3. A generalization $(t_g, \theta_1, \theta_2)$ of t_1 and t_2 is *most specific* (msg) if for every generalization $(t'_g, \theta'_1, \theta'_2)$ of t_1 and t_2 it holds that t_g is an instance of t'_g.
4. Two terms t_1 and t_2 are *disjoint* if their msg is of form (x, θ_1, θ_2).

Algorithm 5 (msg.) An msg $\lfloor s, t \rfloor$ of $s, t \in \mathcal{T}$ is computed by exhaustively applying the rewrite rules in Fig. 10 to the initial triple $(x, \{x := s\}, \{x := t\})$:

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

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

Fig. 10. Computing most specific generalizations.

3.3 Partial Process Trees

A *partial process tree* differs from a process tree in that it may contain an extra kind of nodes, *generalization-nodes*, with label of form **let** $x_1{=}t_1 \ldots x_n{=}t_n$ **in** t' and $n{+}1$ children labeled $t_1, \ldots, t_n, t$, respectively, where $x_1, \ldots, x_n$ do not occur in $t_1, \ldots, t_n$. This kind of node has the distinct feature that the $n + 1$'st edge may go to an ancestor of the node instead of going to a child; such an edge is called a *return edge*. We regard a partial process tree as an acyclic graph by ignoring return edges, so *ancestor, leaf, etc.* apply only to non-return edges. The labels on generalization nodes are unrelated to all other labels wrt. $\trianglelefteq$.

The following definition is inspired by [51].

Definition 6. Let T be a partial process tree with node t with ancestor s.

1. If t is an instance of s, *i.e.* $t \equiv s\{x_1{:=}t_1, \ldots, x_n{:=}t_n\}$, then *FOLD*($T$, s, t) is the tree obtained as follows. Replace t by **let** $x_1{=}t_1 \ldots x_n{=}t_n$ **in** s which is marked, has return edge to s, and n unmarked children $t_1, \ldots, t_n$.
2. If $\lfloor s, t \rfloor = (t_g, \{x_1{:=}t_1, \ldots, x_n{:=}t_n\}, \theta)$, then *GENERALIZE*($T$, s, t) is the partial process tree obtained as follows. Delete all descendants of s, and replace s by **let** $x_1{=}t_1 \ldots x_n{=}t_n$ **in** t_g with a mark and $n + 1$ unmarked children $t_1, \ldots, t_n, t_g$. Return edges from s or its descendants are erased.
3. If $t \equiv h\,t_1 \ldots t_n$ then *SPLIT*(T, s, t) is the partial process tree obtained as follows. Let $t_g \equiv h\; x_1, \ldots, x_n$ where $x_1, \ldots, x_n$ are new variables, replace t by **let** $x_1{=}t_1 \ldots x_n{=}t_n$ **in** t_g which has a mark and $n{+}1$ unmarked children $t_1, \ldots, t_n, t_g$.

Algorithm 7 (positive supercompilation.)

1. INPUT $t_0 \in \mathcal{T}$, $q \in \mathcal{Q}$
2. LET T_0 be the partial process tree with 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 an 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 IF N and M are disjoint THEN $T_{i+1} := \mathit{SPLIT}(T_i, M, N)$
      ELSE $T_{i+1} := \mathit{GENERALIZE}(T_i, M, N)$
   (b) SET $i := i + 1$
4. OUTPUT T_i

The following is a consequence of Kruskal's Tree Theorem, see [14].

Theorem 8. *Algorithm 7 always terminates.*

As for correctness, it is easily proved that each step of the transformation rules preserves normal-order graph reduction semantics; extending rigorously the proof to account for folding is more involved. A general technique due to Sands [60] can be used to prove this for (positive) supercompilation, see [59].

3.4 Discussion of the algorithm

A number of choices are left open or settled in an arbitrary way in our algorithm.

First, our algorithm follows Turchin's *generalization principle* [77] which states that a generalization between two terms has a meaning only in the context of the computation process in which they take part. Indeed, our algorithm searches only the ancestors of a leaf node. However, to avoid the generation of duplicate definitions one might imagine searching across different branches; see *e.g.* [33].

Second, our algorithm does not specify a particular strategy for *selecting unmarked leaf nodes*. One may chose a breath-first or depth-first strategy.

Third, in case driving stops the algorithm may employ different strategies for *selecting ancestors* for generalization. For instance, one may choose the closest ancestor, or the ancestor that gives the most specific generalization.

Fourth, when we perform a $\mathit{GENERALIZE}(T, M, N)$ step we replace node M. Instead one could replace N, since this avoids destroying the whole subtree with root M; other branches from M can be retained with no loss of information.

Fifth, the operator $\lfloor \bullet, \bullet \rfloor$ and the stop criterion can be varied; *e.g.* [47]. The operation $\mathit{SPLIT}(T, s, t)$ may be refined to split t in another way; *e.g.* if $s \equiv h\ x$ and $t \equiv l\ (k\ (h\ y))$ then split such that $t_g \equiv l\ (k\ z)$ (Turchin's algorithm [77] maintains a stack structure of common contexts to determine split points).

Finally, one can imagine various optimizations of which we will discuss only one, namely *transient reductions*. A term of form $e[g\, x\, t_1 \ldots t_n]$ is *non-transient*, all other terms are *transient*. The optimization consists in adding the disjunction "or the label of N is transient" to the condition in (3a) of Algorithm 7. So only terms that involve a choice at run-time are compared to ancestors for whistling in the partial process tree. The rationale is that any loop in the program must pass through a choice point 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 [38]. The partial process tree for $a\ (a\ xs\ ys)\ xs$ using transient reductions appears in Fig. 11.

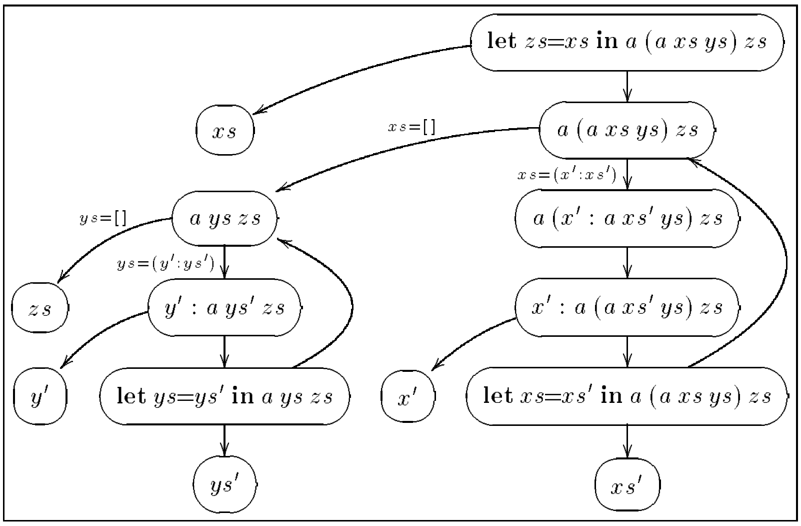


Fig. 11. Example partial process tree with transient reductions.

From this tree one can generate the term $a'\ xs\ ys\ xs$ and a new program (Fig. 12). This is noteworthy because the initial term requires passing the list xs twice, whereas the new term passes xs only once.

$$
\begin{array}{ll}
a'\ [\,]\ ys\ zs & \doteq a''\ ys\ zs \\
a'\ (x : xs)\ ys\ zs & \doteq x : a'\ xs\ ys\ zs \\
 & \\
a''\ [\,]\ zs & \doteq zs \\
a''\ (y : ys)\ zs & \doteq y : a''\ ys\ zs
\end{array}
$$

Fig. 12. More efficient double append program.

3.5 Comparative Remarks

From its very inception, supercompilation has been tied to a specific programming language, called *Refal* [67], a language inspired by Markov algorithms. A Refal program is a sequence of rewrite rules, used to transform data in the form of associative and possibly nested symbol strings and offers certain advantages for programming, *e.g.* [78, 53]. Running interpreters were available by the end of the 1960's; different versions of the language were implemented [83, 43, 78, 35].

The equivalence transformations necessary for supercompilation of Refal, including the central *rule of driving* and the *outside-in strategy*, were formulated in 1972 [68, 69].

Driving and generalization for our language are simplified considerably due to simpler data structures, untyped variables, and flat patterns (essentially *elementary contractions* [77]). Due to Refal's data structure most general unifiers do not always exist; a *generalized matching algorithm* is needed [44, 72, 75].

We should note that supercompilation, as defined by Turchin, is a normal-order transformation that is applied to programs with call-by-value semantics, and that transformed programs are again interpreted call-by-value. As a result, supercompilation may make programs terminate more often. The positive supercompiler defined here transforms programs with normal-order graph reduction semantics into programs with the same semantics and the same termination properties.

Process trees correspond to Turchin's *graph of states* [73], sometimes called Refal graphs. A version of driving was used in the seventies in a system for interpretive inversion, called URA; *c.f.* [55] (see Sect. 5.2). Several supercompilers have been developed for Refal [86, 57, 42, 77, 34, 85]; the first 'non-Refal' supercompiler was [29]. Driving has been used for *neighborhood analysis* [72] to determine sets of data that pass, up to a certain point, through a computation process in identical ways; the use of neighborhoods has been suggested for generalization [77] and program testing [2, 3].

4 Related Program Transformers

In this section we compare positive supercompilation briefly to *partial evaluation*, *deforestation*, *partial deduction*, *perfect supercompilation*, and *generalized partial computation*. First we introduce a number of axes along which transformers can be compared, and then enter the coordinates of the above transformers.

4.1 Some Dimensions in Automatic Program Transformation

Information propagation. Every program transformer maintains a certain level of information propagation; we consider *constant propagation*, *unification-based information propagation*, and *constraint-based information propagation*.

The three levels differ in how much information is recorded about pattern matching and tests, corresponding to the transformation rules in Fig. 13.

	$T[\![$ **if** $u{=}v$ **then** t **else** s $]\!] =$	information propagation
(a)	**if** $u{=}v$ **then** $T[\![\ t\ ]\!]$ **else** $T[\![\ s\ ]\!]$	constant propagation
(b)	**if** $u{=}v$ **then** $T[\![\ t\{u{:=}v\}\ ]\!]$ **else** $T[\![\ s\ ]\!]$	unification-based
(c)	**if** $u{=}v$ **then** $T[\![\ t\ ]\!]\{u = v\}$ **else** $T[\![\ s\ ]\!]\{u \neq v\}$	constraint-based

Fig. 13. Information propagation.

In constant propagation the outcome of tests are ignored. In unification-based propagation substitutions into the transformed terms are used to represent the outcome of tests. In constraint-based propagation the transformer explicitly maintains sets of constraints recording previous tests (*restrictions* [72, 29]). Depending on the programming language other abstract properties may be propagated, *e.g.* [61, 13, 66, 37].

Evaluation strategy. One can view a program transformer as an extension of an interpreter, *e.g.* [29, 27, 52, 65]. This implies that the transformer has an *evaluation strategy* that it inherits from the underlying interpreter. More concretely, the transformer processes nedsted function calls in some order. We consider transformers that use *inside-out* (or *call-by-value* or *applicative order*) and *outside-in* (or *call-by-name* or *normal-order*).

Control restructuring. Control restructuring is concerned with the relationship between program points in the subject and the residual program [9, 58]:

Monovariant: any program point in the subject program gives rise to zero or one program point in the residual program.
Polyvariant: any program point in the subject program can give rise to one or more program points in the residual program.
Monogenetic: any program point in the residual program is produced from a single program point of the subject program.
Polygenetic: any program point in the residual program may be produced from one or more program points of the subject program.

4.2 A Taxonomy of Transformers

Deforestation, due to Wadler [87], performs *program composition* by eliminating intermediate data structures. Deforestation performs, as a special case, program specialization [64]. Deforestation is very similar to positive supercompilation except that it uses constant propagation rather than unification-based information propagation, and it does not incorporate generalization; instead it is guaranteed to terminate for a certain class of programs.

Partial evaluation performs *program specialization* and, as presented in [38], uses only constant propagation [29, 64, 65]. This limitation applies to all variants of partial evaluation: offline and online approaches with and without partially static structures. The usual evaluation strategy for partial evaluators is applicative-order, see [52].

Partial deduction, as in [48, 45, 22], and positive supercompilation have essential aspects in common [33]: the way in which goals are unified and how the resulting substitutions are applied to the goals in the next transformation step (construction of a partial SLDNF tree), is much like in the clauses of driving.

Since in logic programs predicates cannot occur inside predicates, there is no direct correspondence to the rules for nested function calls which achieve deforestation. However, local variables in logic programs often represent intermediate data structures that could be removed by more sophisticated techniques. Partial deduction in logic programming is not capable of removing them; this requires an extension of the techniques, see *e.g.* [54].

Turchin's supercompiler [75] and our positive supercompiler are identical with respect to the propagation of positive information, except for certain trivial differences. The main difference between the two is that the former also maintains *negative information*, *i.e.* the information that a test failed, and this is maintained in the form of constraints (see *perfect driving* [29]).

Generalized partial computation (GPC), due to Futamura [21], has a similar effect and power as supercompilation, but has arbitrary tests in conditionals rather than just equality tests. The underlying logic for the tests can be any logic system, for example predicate logic, and may be undecidable for certain logic formulas. In this view, positive supercompilation can be seen as propagating structural predicates that can always be resolved.

These observations are summarized in Fig. 14. For a more detailed discussion on information propagation see [29, 64, 33, 65], and for more on evaluation strategies see [11, 52]. These papers also give examples of optimizations that require the transformer to use a specific evaluation strategy or level of information propagation. For instance, to pass the so-called *KMP-test* [64], at least unification-based propagation is required; to eliminate intermediate data structures in general, normal-order strategy is required.

transformer	information propagation	evaluation strategy	control restruct.		KMP test	data struct.
			variant	genetic		
Partial evaluation	constant	in-out	poly	mono	–	–
Deforestation	constant	out-in	poly	poly	–	+
Partial deduction	unification	unspecified	poly	mono	+	–
Positive SCP	unification	out-in	poly	poly	+	+
Perfect SCP	constraint	out-in	poly	poly	+	+
GPC	constraint	out-in	poly	poly	+	+

Fig. 14. A taxonomy of transformers.

5 Larger Perspectives of Supercompilation

Supercompilation achieves program specialization, but is not limited to this application: it is a much wider framework for equivalence transformation of programs. Program inversion is one of the more advanced applications of supercompilation which we will outline in this section.

We refer to any process of simulating, analyzing or transforming programs by means of programs as *metacomputation*; the term stresses the fact that this activity is one level higher than ordinary computation ("programs as data objects"). Program specialization, composition, and inversion are different metacomputation tasks; programs that carry out these tasks, are *metaprograms*. The step from a program to the application of a metaprogram to the encoded form of the program is a *metasystem transition*; repeated use of metasystem transition leads to a *multi-level metasystem hierarchy*. We adopt a language-independent formalization [26] based on [72, 23, 30, 84].

Metasystem transition is a key ingredient of Turchin's approach [71]: the construction of hierarchies of metasystems (*e.g.* supercompilers) was taken as the basis for program analysis and transformation. The book [83] defined all three Futamura projections in terms of metasystem transition.

Sect. 5.1 introduces a formalism for metacomputation, in Sect. 5.2 discusses supercompilation and program inversion, and Sect. 5.3 presents metasystem transition.

5.1 Metacomputation Revisited

Computation. We assume a fixed set D in which programs written in different languages, as well as their input and output data, are members. To express the application of programs to data we define an *application language* A by the grammar

$$A \ ::= \ D \ | \ \texttt{<}A \ \ A^*\texttt{>}$$

where the symbols $\texttt{<},\texttt{>} \notin D$ denote the application of a program to its inputs. Capitalized names in `typewriter` font denote arbitrary elements of D. They are free variables of the meta-notation in which the paper is written. For instance, the intended meaning of the A-expression $\texttt{<P X Y>}$ is the application of program $\texttt{P} \in D$ to the input $\texttt{X}, \texttt{Y} \in D$.

We are not interested in a specific programming language for writing programs. For simplicity, let all source-, target- and metalanguages be identical.

We write $a \Rightarrow \texttt{D}$ to denote the *computation* of an expression $a \in A$ to $\texttt{D} \in D$. For instance, $\texttt{<P X Y>} \Rightarrow \texttt{OUT}$ is the computation of program $\texttt{P} \in D$ with inputs $\texttt{X}, \texttt{Y} \in D$ and output $\texttt{OUT} \in D$. Two A-expressions $a, b \in A$ are computationally equal if they can be reduced to identical D-expressions:

$$a = b \ \ \textit{iff} \ \ \forall \texttt{X} \in D\colon (a \Rightarrow \texttt{X} \ \ \textit{iff} \ \ b \Rightarrow \texttt{X})$$

Abstraction. To represent sets of A-expressions, we define a *metacomputation language* B by the grammar

$$B \ ::= \ D \ | \ M \ | \ \texttt{<}B \ \ B^*\texttt{>}$$

where M is a set of *metavariables*. A metavariable $\texttt{m} \in M$ is a placeholder that stands for an unspecified data element $\texttt{D} \in D$. We use lowercase names in `typewriter` font to write elements of M. A B-expression b is an abstraction that represents the set of all A-expressions obtained by replacing metavariables $\texttt{m} \in M$ by elements of D. We write $a \in b$ to denote that $a \in A$ is an element of the set represented by $b \in B$. We refer to a B-expression also as a *configuration*.

Encoding. Expressions in the metacomputation language need to be represented as data in order to manipulate them by means of programs (ordinary computation cannot reduce B-expressions because metavariables are not in A). A *metacoding* [72] is an injective mapping $B \to D$ to encode B-expressions in D. We are not interested in a specific way of metacoding and assume some metacoding $\overline{\bullet} : B \to D$. Repeated metacoding is well-defined because $D \subset B$.

Metacomputation. It follows from our notation that $\texttt{<MC }\overline{b}\texttt{>} \Rightarrow \texttt{D}$ denotes metacomputation on an expression $b \in B$ using a metaprogram $\texttt{MC} \in D$. The application of `MC` to the metacoded B-expression is an A-expression that can be reduced by ordinary computation. We should stress that this characterization of metacomputation says nothing about its concrete nature, except that it involves a metaprogram `MC` that operates on a metacoded configuration $\overline{b}$. Different metaprograms may perform different operations on b, such as program specialization, program composition, or program inversion.

Definition 9 (program inverter). A program $\texttt{INV} \in D$ is a *program inverter* if for every program $\texttt{P} \in D$ injective in its first argument,[4] every input $\texttt{X}, \texttt{Y} \in D$ and metavariable $\texttt{x} \in M$, there exists a program $\texttt{P}^{-1} \in D$ such that

$$\texttt{<INV }\overline{\texttt{<P x Y>}}\texttt{>} \Rightarrow \texttt{P}^{-1} \quad \text{and} \quad \texttt{<P}^{-1}\texttt{ <P X Y>>} \Rightarrow \texttt{X}$$

In general when `P` is not injective, $\texttt{P}^{-1}$ must return a list of results.[5]

5.2 Interpretive Inversion by Supercompilation

Supercompilation is capable of *interpretive inversion* [69, 55, 1] (we show later how metasystem transition can be used to generate an inverse program $\texttt{P}^{-1}$). The formulation of interpretive inversion is as follows. Let `EQ` be a program that tests the equality of two data elements. Given $\texttt{Y}, \texttt{Z}$ find an `X` such that

$$\texttt{<EQ <P X Y> Z>} \Rightarrow \texttt{'True'}$$

where `'True'` is some distinct element of D. Supercompilation, more specifically driving, can be used to obtain a program `ANSWER` with answers for `x` internalized:

$$\texttt{<DRIVE }\overline{\texttt{<EQ <P x Y> Z>}}\texttt{>} \Rightarrow \texttt{ANSWER}$$

Example 1. Let numbers be represented by lists of length n. Then program append a (Fig. 2) implements the addition of two numbers. Using driving (Sect. 2) we can compute $z - y$ by interpretive inversion of addition. The result of interpretive inversion for $z = 1$ and $y = 0$, *i.e.* driving eq (a xs [ ]) [1], appears in Fig. 15. The answer, $x = 1$, can be extracted mechanically from the program.

[4] `P` is injective in its first argument if for all $\texttt{X1}, \texttt{X2}, \texttt{Y} \in D$: $\texttt{<P X1 Y>} = \texttt{<P X2 Y>}$ implies that `X1` and `X2` are the same element of D.

[5] There are two types of inversions: either we are interested in an *existential* solution (one of the possible results), or in a *universal* solution (all possible results).

$$\begin{array}{|llll|}\hline g_1\ [\,] & \doteq \mathit{False} & g_2\ [\,] & \doteq \mathit{True} \\ g_1\ (x\!:\!xs) & \doteq g_2\ xs & g_2\ (x\!:\!xs) & \doteq \mathit{False} \\ \hline \end{array}$$

Fig. 15. Result of driving $eq\ (a\ \ xs\ \ [\,])\ [1]$.

Example 2. Using supercompilation instead of driving one may produce a finite program even when the list of possible answers is infinite. This may be used for theorem proving [74, 86]. An example is shown in Fig. 16 where the supercompiler (Sect. 3) is applied to $eq\ (a\ \ xs\ \ [\,])\ xs$ which represents the proposition $\forall n.(n + 0 = n)$ which can be proven only by using induction. The residual program constructed returns *True* for all lists. This proves the theorem.

$$\begin{array}{|ll|}\hline g_1\ [\,] & \doteq \mathit{True} \\ g_1\ (x\!:\!xs) & \doteq g_1\ xs \\ \hline \end{array}$$

Fig. 16. Result of supercompiling $eq\ (a\ \ xs\ \ [\,])\ xs$.

One of the first results for interpretive inversion by driving were obtained in 1972 by performing subtraction by interpretive inversion of binary addition [68]. In 1973 S.A. Romanenko and later S.M. Abramov implemented an algorithm, Universal Resolving Algorithm (URA), in which driving was combined with a mechanical extraction of answers, see [55]. For program inversion see also [55, 36, 56, 81]. The generation of an algorithm representing binary subtraction from binary addition by self-application was reported in [34].

In logic programming, one defines a predicate by a program <P x y> and solves the inversion problem for Z = 'True'. Theorem proving and program transformation are indistinguishable in the approach outlined above; they are two applications of the same equivalence transformation. The definition of a predicate may be perceived as non-procedural, but their semantics is still defined in terms of computation. The application of supercompilation to problem solving and theorem proving has been discussed in [74, 75], the connection to logic programming in [1, 24, 33].

5.3 Metasystem Transition

Having introduced the basic concepts of metacomputation, we now consider the use of multi-level metasystem hierarchies together with a supercompiler. During the construction of multi-level hierarchies, we will frequently need to replace metacoded subexpressions by metavariables. The correct treatment of metacode is so essential in self-application [23], that we make elevated metavariables [84] an integral part of the MST-language. We define a *metasystem transition language* C by the grammar

$$C\ ::=\ D \mid M_{I\!N} \mid \texttt{<}C\ \ C^{*}\texttt{>}$$

where $M_{I\!N}$ is a set of *elevated metavariables* $\mathtt{m_H}, \mathtt{H} \in I\!N$. An elevated metavariable $\mathtt{m_H}$ ranges over data metacoded H-times. We will denote by $D^{\mathtt{H}}$ the set of metacode $\overline{\mathtt{D}}^{\mathtt{H}}$ of all $\mathtt{D} \in D$. A metavariable without elevation has `0` as its elevation index. A C-expression c represents the set of all A-expressions obtained by substituting elevated metavariables $\mathtt{m}_{\mathtt{H}}$ by elements of $D^{\mathtt{H}}$.

Metasystem Transition. The construction of each next level in a metasystem hierarchy, referred to as a *metasystem transition* (MST) [83], is done in three steps [31]:

(A) given an initial A-expression a,
(B) define a C-expression c such that $a \in c$,
(C) apply a metaprogram `MC` to the metacode $\overline{c}$.

The expression obtained in the last step is again an A-expression and the same procedure can be repeated. Expressions obtained by MST are called *MST-formulas*. This definition says nothing about the goal of the MST, except that it is an abstraction of an A-expression a to a configuration c, followed by the application of a metaprogram `MC` to $\overline{c}$.

Generating Inverse Programs. The interpretive inversion of a program can always be performed using driving, but the performance can be poor whilst often more efficient inverse programs are known to exist. Figure 17 show how MST can be used to synthesize inverse programs by specialization of the universal resolving algorithm `URA`; see [55, 1]. For notational convenience let `<Q x y z>` be defined by `<EQ <P x y> z>`. A specializer `SPEC` is used for the sake of generality, but it should be clear that a supercompiler `SCP` can be used instead.

- **1st MST** Define a C-expression (B0) by replacing `X` by $\mathtt{x}_0$ in the A-expression (A0), and apply `URA` to the metacoded C-expression (A1) to perform interpretive inversion: the 1st MST. Interpretive inversion of `Q` is achieved.
- **2nd MST** Define a C-expression (B1) by replacing $\overline{\mathtt{Y}}, \overline{\mathtt{Z}}$ by $\mathtt{y}_1, \mathtt{z}_1$ in the A-expression (A1)[6], and apply $\mathtt{SPEC}'$ to the metacoded C-expression (A2) to specialize `URA` and remove its interpretive overhead: the 2nd MST. The result is an inverted program $\mathtt{Q}^{-1}$ that returns `ANSWER` given `Y`, `Z`.
- **3rd MST** Define a C-expression (B2) by replacing $\overline{\overline{\mathtt{Q}}}$ by `q2` in the A-expression (A2), and apply $\mathtt{SPEC}''$ to the metacoded C-expression (A3): the 3rd MST. The result is an inverter `INV` that converts a program `Q` into $\mathtt{Q}^{-1}$.
- **4th MST** Define a C-expression (B3) by replacing $\overline{\overline{\overline{\mathtt{URA}}}}$ by $\mathtt{ura}_2$ in the A-expression (A3), and apply $\mathtt{SPEC}'''$ to the metacoded C-expression (A4): the 4th MST. The result is an inverter generator `INVGEN`.

A hierarchy of metasystems can be visualized using a *2-dimensional notation*[7]: (i) an expression is moved down one line down for each metacoding; (ii) the elevation of a metavariable $\mathtt{m}_{\mathtt{H}}$ is shown by a bullet • located `H` lines below the

[6] We take the liberty to replace subexpressions of $d \in D$ by metavariables and interrupt the horizontal line above the enclosing expression; defined formally in [26].
[7] Introduced by Turchin; a preliminary form appeared in [23].

(A0)	$\texttt{<Q X Y Z>} \Rightarrow \texttt{BOOL}$	(computation)
(B0)	$\texttt{<Q x}_0\texttt{ Y Z>}$	(abstraction)
(A1)	$\texttt{<URA }\overline{\texttt{<Q x}_0\texttt{ Y Z>}}\texttt{>} \Rightarrow \texttt{ANSWER}$	(1st MST)
(B1)	$\texttt{<URA }\overline{\texttt{<Q x}_0}\texttt{ y}_1\texttt{ z}_1\overline{\texttt{>}}\texttt{>}$	(abstraction)
(A2)	$\texttt{<SPEC}'\ \overline{\texttt{<URA }\overline{\texttt{<Q x}_0}\texttt{ y}_1\texttt{ z}_1\overline{\texttt{>}}\texttt{>}}\texttt{>} \Rightarrow \texttt{Q}^{-1}$	(2nd MST)
(B2)	$\texttt{<SPEC}'\ \overline{\texttt{<URA }\overline{\texttt{<}}}\texttt{q}_2\ \overline{\overline{\texttt{x}_0}\texttt{ y}_1\texttt{ z}_1\overline{\texttt{>}}}\texttt{>>}$	(abstraction)
(A3)	$\texttt{<SPEC}''\ \overline{\texttt{<SPEC}'\ \overline{\texttt{<URA }\overline{\texttt{<}}}\texttt{q}_2\ \overline{\overline{\texttt{x}_0}\texttt{ y}_1\texttt{ z}_1\overline{\texttt{>}}}\texttt{>>}}\texttt{>} \Rightarrow \texttt{INV}$	(3rd MST)
(B3)	$\texttt{<SPEC}''\ \overline{\texttt{<SPEC}'\ \overline{\texttt{<}}}\texttt{ura}_2\ \overline{\overline{\overline{\texttt{<}}}\texttt{q}_2\ \overline{\overline{\texttt{x}_0}\texttt{ y}_1\texttt{ z}_1\overline{\texttt{>}}}}\texttt{>>>}$	(abstraction)
(A4)	$\texttt{<SPEC}'''\ \overline{\texttt{<SPEC}''\ \overline{\texttt{<SPEC}'\ \overline{\texttt{<}}}\texttt{ura}_2\ \overline{\overline{\overline{\texttt{<}}}\texttt{q}_2\ \overline{\overline{\texttt{x}_0}\texttt{ y}_1\texttt{ z}_1\overline{\texttt{>}}}}\texttt{>>>}}\texttt{>} \Rightarrow \texttt{INVGEN}$	(4th MST)

Fig. 17. MST-formulas for program inversion.

metavariable. The two-dimensional version of the 3rd MST (A3) is shown below.

```
<SPEC''_________________________> ⇒ INV
       <SPEC'_____ q ___________>
              <URA_ | __y z __>
                   < • x • • >
```

The *Futamura projections* [19] are, in all probability, the first example of program transformation beyond a single metasystem level, but MST is not limited to this application. Turchin suggested MST to increase the power of theorem proving [74] and it inspired a constructive approach to the foundations of mathematics [76]. The philosophical background of MST was exposed in [70]; see also [79, 80].

First successful self-application of a partial evaluator was achieved by Jones, Sestoft and Søndergaard [39]. The generation of an algorithm representing binary subtraction from binary addition by self-application was reported in [34]. Examples of MST are multiple self-application [23, 28], the generation of program transformers [25, 27], and other related applications in [42, 1, 24, 31, 32].

Acknowledgments. Thanks to Sergei Abramov, Andrei Klimov, Andrei Nemytykh, Alexander Romanenko (†), Sergei Romanenko, and last but not least, Valentin F. Turchin for many interesting discussions and hospitality during visits in Russia and New York. Special thanks are due to Andrei Klimov for transcription and translation of Russian titles. Thanks to John Hatcliff, Neil D. Jones, Jesper Jørgensen, Bern Martens, Kristian Nielsen, and David Sands for stimulating discussions on various topics of this paper. Finally, we are indebted to all members of the Topps group at DIKU for providing an excellent working environment.

References

1. S.M. Abramov. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31–44, 1991. (In Russian).
2. S.M. Abramov. Metacomputation and program testing. In *1st International Workshop on Automated and Algorithmic Debugging (Linköping, Sweden)*, pages 121–135, 1993.
3. S.M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications)*. Nauka, Moscow, 1995. (In Russian).
4. M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. Manuscript, Universidad Politecnica de Valencia, Spain, 1995.
5. L. Beckman, A. Haraldson, Ö. Oskarsson, and E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.
6. R. Bird and P.L Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.
7. D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.
8. R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
9. M.A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.
10. C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics. Academic Press, 1973.
11. C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 495–519. Springer-Verlag, 1991.
12. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium in Principles of Programming Languages*, pages 493–501. ACM Press, 1993.
13. C. Consel and S.C. Khoo. Parameterized partial evaluation. *ACM TOPLAS*, 15(3):463–493, 1993.
14. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1992.
15. J. Dixon. The specializer, a method of automatically writing computer programs. Technical report, Division of Computer Research and Technology, National Institute of Health, Bethesda, Maryland, 1971.
16. A.P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.
17. A.P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
18. A. Ferguson and P.L. Wadler. When will deforestation stop? In *1988 Glasgow Workshop on Functional Programming*, pages 39–56, 1988.
19. Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
20. Y. Futamura. Partial computation of programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *RIMS Symposia on Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 1–35, Kyoto, Japan, 1983. Springer-Verlag.
21. Y. Futamura. Program evaluation and generalized partial computation. In *International Conference on Fifth Generation Computer Systems*, pages 1–8, 1988.

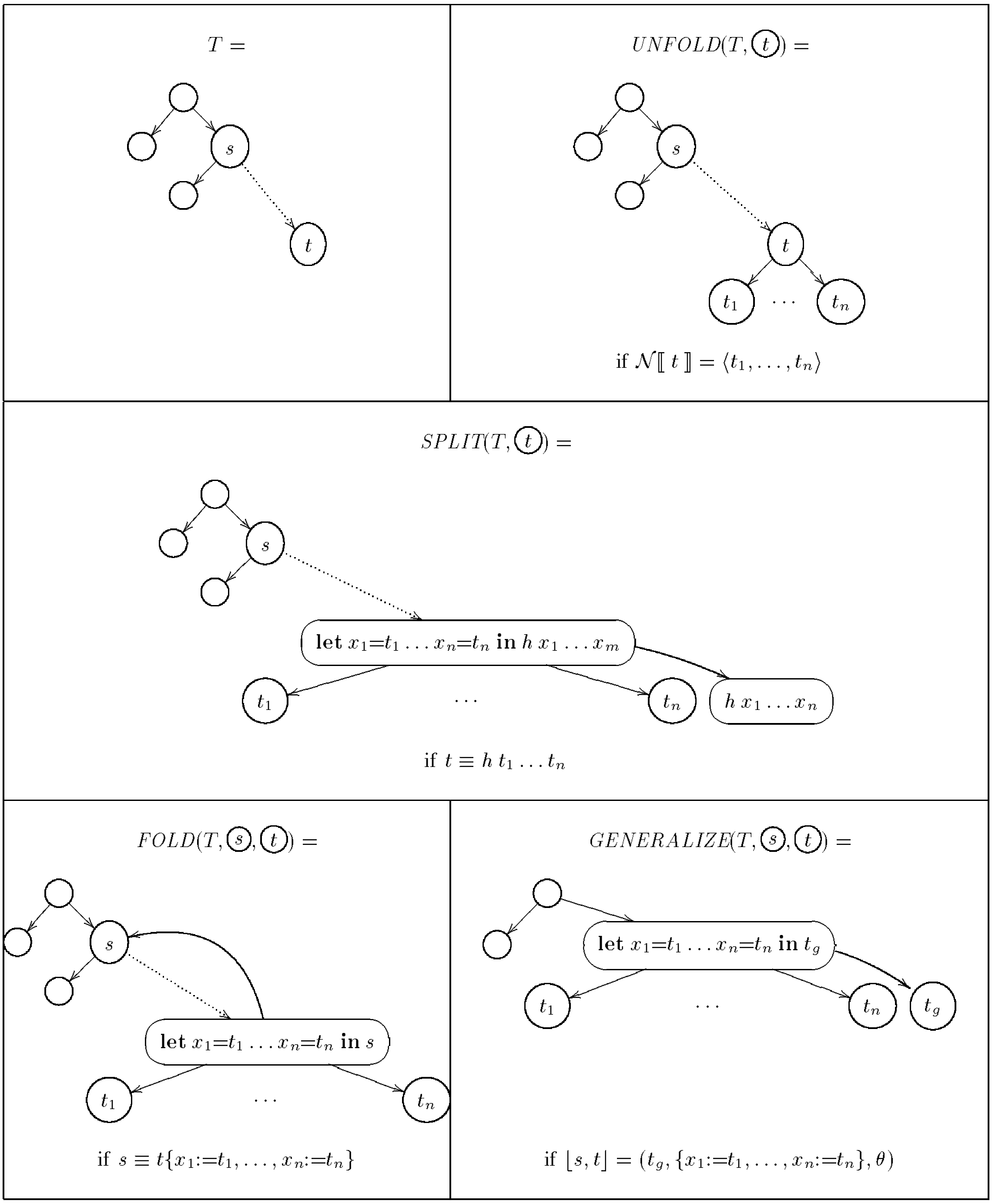
22. J. Gallagher. Tutorial in specialisation of logic programs. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
23. R. Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320, New Haven, Connecticut, 1991. ACM Press.
24. R. Glück. Projections for knowledge based systems. In R. Trappl, editor, *Cybernetics and Systems Research'92. Vol. 1*, pages 535–542. World Scientific: Singapore, 1992.
25. R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, 1994.
26. R. Glück. On the mechanics of metasystem hierarchies in program transformation. page submitted, 1995.
27. R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis. Proceedings*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448, Namur, Belgium, 1994. Springer-Verlag.
28. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S.D. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.
29. 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, *Static Analysis. Proceedings. Lecture Notes in Computer Science, Vol. 724*, pages 112–123. Springer-Verlag, 1993.
30. R. Glück and A.V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl, editor, *Cybernetics and Systems '94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.
31. R. Glück and A.V. Klimov. Metasystem transition schemes in computer science and mathematics. *World Futures: the Journal of General Evolution*, page (to appear), 1995.
32. R. Glück and A.V. Klimov. Reduction of language hierarchies. In *Proceedings of the 14th International Congress on Cybernetics*, page to appear, Namur, Belgium, 1995. International Association for Cybernetics.
33. R. Glück and M.H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181, Madrid, Spain, 1994. Springer-Verlag.
34. R. Glück and V.F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the ISSAC'90 (Tokyo, Japan)*, pages 286–287. ACM Press, 1990.
35. R. Gurin and S.A. Romanenko. *Jazyk programmirovanija Refal Pljus (The Refal Plus programming language)*. Intertech, Moscow, 1991.
36. P.G. Harrison. Function inversion. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 153–166, Gammel Avernæs, Denmark, 1988. North-Holland.
37. N.D. Jones. The essence of program transformation by partial evaluation and driving. In N.D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language, and Computation*, volume 792 of *Lecture Notes in Computer Science*, pages 206–224. Springer-Verlag, 1994. Festschrift in honor of S.Takasu.

38. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
39. N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140. Springer-Verlag, 1985.
40. N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
41. S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
42. A.V. Klimov and S.A. Romanenko. Metavychislitel' dlja jazyka Refal. Osnovnye ponjatija i primery. (A metaevaluator for the language Refal. Basic concepts and examples). Preprint 71, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987.
43. A.V. Klimov and S.A. Romanenko. Sistema programmirovanija Refal-2 dlja ES. Opisanie vkhodnogo jazyka (Programming system Refal-2 for ES computers. The source language description). Technical report, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987.
44. A.V. Klimov, S.A. Romanenko, and V.F. Turchin. Teoreticheskie osnovy sintaksicheskogo otozhdestvlenija v jazyke Refal (The theory of pattern matching in Refal). Preprint 13, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1973.
45. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Meta-Programming in Logic*, volume 649 of *Lecture Notes in Computer Science*, pages 49–69, 1992.
46. J. Komorowski. Special issue on partial deduction. *Journal of Logic Programming*, 16(1&2):1–189, 1993.
47. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. 1995. submitted.
48. J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.
49. L.A. Lombardi. Incremental computation. In F. L. Alt and M. Rubinoff, editors, *Advances in Computers*, volume 8, pages 247–333. Academic Press, 1967.
50. L.A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In E.C. Berkeley and D.G. Bobrow, editors, *The Programming Language Lisp: Its Operation and Applications*, pages 204–219, Cambridge, Massachusetts, 1964. MIT Press.
51. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Stirling, editor, *International Conference on Logic Programming*, pages 597–613. MIT Press, 1995.
52. K. Nielsen and M.H. Sørensen. Call-by-name CPS-translation as a binding-time improvement. In A. Mycroft, editor, *Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 296–3131. Springer-Verlag, 1995.
53. R.M. Nirenberg. A practical turing machine representation. *SIGACT News*, 17(3):35–44, 1986.
54. M. Proietti and A. Pettorossi. Unfolding – definition – folding, in this order for avoiding unnecessary variables in logic programs. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 347–358. Springer-Verlag, 1991.

55. A.Y. Romanenko. The generation of inverse functions in Refal. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 427–444, Gammel Avernæs, Denmark, 1988. North-Holland.
56. A.Y. Romanenko. Inversion and metacomputation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (Yale University, Connecticut)*, pages 12–22. ACM Press, 1991.
57. S.A. Romanenko. Progonka dlja programm na Refale-4 (Driving for Refal-4 programs). Preprint 211, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987.
58. S.A. Romanenko. Arity raiser and its use in program specialization. In N.D. Jones, editor, *ESOP'90*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360. Springer-Verlag, 1990.
59. D. Sands. Proving the correctness of recursion-based automatic program transformation. In P. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 681–695. Springer-Verlag, 1995.
60. D. Sands. Total correctness by local improvement in program transformation. In *22nd Symposium on Principles of Programming Languages*, pages 221–232. ACM Press, 1995.
61. D. Smith. Partial evaluation of pattern matching in constraint logic programming. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press, 1991.
62. M.H. Sørensen. Turchin's supercompiler revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.
63. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *International Logic Programming Symposium*. MIT Press, 1995. to appear.
64. M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *Programming Languages and Systems*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1994.
65. M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. Submitted to Journal of Functional Programming, 1995.
66. 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.
67. V.F. Turchin. Metajazyk dlja formal'nogo opisanija algoritmicheskikh jazykov (A metalanguage for the formal description of algorithmic languages). In *Cifrovaja Vychislitel'naja Tekhnika i Programmirovanie*, pages 116–124. Sovetskoe Radio, Moscow, 1966. (In Russian).
68. V.F. Turchin. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal). In *Teorija Jazykov i Metody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods). (Kiev-Alushta, USSR)*, pages 31–42, 1972. (In Russian).
69. V.F. Turchin. Ehkvivalentnye preobrazovanija programm na Refale (Equivalent transformations of Refal programs). In *Avtomatizirovannaja Sistjema upravljenija stroitel'stvom. Trudy TxNIPIASS 6*, pages 36–68, 1974. (In Russian).

70. V.F. Turchin. *The Phenomenon of Science.* Columbia University Press, New York, 1977.
71. V.F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, 1979.
72. V.F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
73. V.F. Turchin. Semantic definitions in Refal and the automatic production of compilers. In N.D. Jones, editor, *Workshop on Semantics-Directed Compiler Generation, Århus, Denmark. Lecture Notes in Computer Science 94*, pages 441–474. Springer-Verlag, January 1980.
74. V.F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J.W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657, Noordwijkerhout, Netherlands, 1980. Springer-Verlag.
75. V.F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
76. V.F. Turchin. A constructive interpretation of the full set theory. *The Journal of Symbolic Logic*, 52(1):172–201, 1987.
77. V.F. Turchin. The algorithm of generalization. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
78. V.F. Turchin. *Refal-5, Programming Guide and Reference Manual.* New England Publishing Co., Holyoke, Massachusetts, 1989.
79. V.F. Turchin. The cybernetic ontology of action. *Kybernetes*, 22(2):10–30, 1993.
80. V.F. Turchin. On cybernetic epistemology. *Systems Research*, 10(1):3–28, 1993.
81. V.F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
82. V.F. Turchin. On generalization of lists and strings in supercompilation. Technical report, City College of the City University of New York, 1995.
83. V.F. Turchin, And. V. Klimov, Ark. V. Klimov, V.F. Khoroshevsky, A.G. Krasovsky, S.A. Romanenko, I.B. Shchenkov, and E.V. Travkina. *Bazisnyj Refal i ego realizacija na vychislitelnykh mashinakh (Basic Refal and its implementation on computers).* GOSSTROJ SSSR, CNIPIASS, Moscow, 1977.
84. V.F. Turchin and A.P. Nemytykh. Metavariables: their implementation and use in program transformation. Technical Report CSc. TR 95-012, City College of the City University of New York, 1995.
85. V.F. Turchin and A.P. Nemytykh. A self-applicable supercompiler. Technical Report CSc. TR 95-010, City College of the City University of New York, 1995.
86. V.F. Turchin, R. Nirenberg, and D. Turchin. Experiments with a supercompiler. In *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pages 47–55. ACM Press, 1982.
87. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88 LNCS vol. 300.

A Operations on Partial Process Trees



This article was processed using the LaTeX macro package with LLNCS style