In: Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 1996, Lecture Notes in Computer Science 1110, Springer, 1996 pp. 355-385. (Invited paper)

A Comparative Revisitation of Some Program Transformation Techniques

Alberto Pettorossi 1 and Maurizio Proietti 2

1 Department of Informatics, Systems, and Production,
University of Roma Tor Vergata, 00133 Roma, Italy. adp@iasi.rm.cnr.it
2 IASI-CNR, Viale Manzoni 30, 00185 Roma, Italy. proietti@iasi.rm.cnr.it

Abstract. We revisit the main techniques of program transformation which are used in partial evaluation, mixed computation, supercompilation, generalized partial computation, rule-based program derivation, program specialization, compiling control, and the like. We present a methodology which underlines these techniques as a 'common pattern of reasoning' and explains the various correspondences which can be established among them. This methodology consists of three steps: i) symbolic computation, ii) search for regularities, and iii) program extraction. We also discuss some control issues which occur when performing these steps.

1 Introduction

During the past years researchers working in various areas of program transformation, such as partial evaluation, mixed computation, supercompilation, generalized partial computation, rule-based program derivation, program specialization, and compiling control, have been using very similar techniques for the development and derivation of programs.

Unfortunately, that similarity has not always been given enough attention because of some lack of interaction among the various groups of researchers involved in these areas. This was motivated by the fact that the objectives of these groups were somewhat different, as for instance, program derivation, compiler generation, and program optimization. Another reason for the lack of interaction was the fact that the programming languages used, whether imperative, functional, or logic, often made a significant difference in the way the various techniques were actually implemented and applied.

In recent years comparisons have been made and correspondences have been established among the different techniques in some particular cases [25, 44, 48]. For some time already, the scientific community has been aware that many such correspondences exist in general, and they are based on the fact that those techniques all share the same underlining methodology which we want to describe in this paper. This general methodology shows that correspondence results may have somewhat complex formalizations, but they are not accidental.

We know from various papers, conferences, and discussions with people working in the area of program transformation that the methodology we will describe here is indeed 'common knowledge'. Thus, the aim of this work is mainly to clarify some issues related to this common knowledge and, as a side-effect, to indicate why the correspondence results do hold and also to present the main features of a general framework where different transformation techniques could be combined together.

2 A Preliminary Example

In this section we revisit a familiar example of program derivation using a functional language based on first-order recursive equations and the *unfold/fold* transformation system with rules and strategies [10, 40]. This revisitation allows us to present in a concrete case the three steps of the general methodology for program transformation we want to introduce, namely, i) symbolic computation, ii) search for regularities, and iii) program extraction. Various instances of this methodology were developed in the seventies independently by many people in several research fields such as partial evaluation, mixed computation, unfold/fold transformation, and supercompilation.

Suppose we are given the following initial program for computing the Fibonacci function:

1. $\mathit{fib}(0) = 1$
2. $\mathit{fib}(1) = 1$
3. $\mathit{fib}(n+2) = \mathit{fib}(n+1) + \mathit{fib}(n)$ for $n \geq 0$

The computation of $\mathit{fib}(k)$ for any natural number $k \geq 0$, requires an exponential number of sums. We want to derive a more efficient program so that the number of the required sums is at most linear for *all* $k \geq 0$. This universal quantification of the variable k over the set of natural numbers, motivates the first step of the general methodology which consists in considering a single *symbolic computation* depending on k (or possibly a finite set of symbolic computations), instead of the infinite set of concrete computations, one for each value of k.

Various models of symbolic computations have been proposed in the literature within various program transformation systems. We will consider here the m-dag model [3] which given a recursive program, uses a directed acyclic graph to represent the father-son relationship among the function calls evoked by the given program.

Thus, in our case, starting from the root-node $\mathit{fib}(k)$ we generate, using Equation 3, the two son nodes $\mathit{fib}(k{-}1)$ and $\mathit{fib}(k{-}2)$. The arguments $k{-}1$ and $k{-}2$ are computed by *matching* in the algebra of integers. Thus, for instance, $\mathit{fib}(k)$ which matches the left-hand-side $\mathit{fib}(n{+}2)$ of Equation 3 for $n = k{-}2$ evokes the two recursive calls $\mathit{fib}(k{-}1)$ and $\mathit{fib}(k{-}2)$, corresponding to $\mathit{fib}(n{+}1)$ and $\mathit{fib}(n)$, respectively, in the right-hand-side of that equation. From the node $\mathit{fib}(k-1)$ we then generate the son nodes $\mathit{fib}(k-2)$ and $\mathit{fib}(k-3)$, and we identify the two distinct nodes for $\mathit{fib}(k{-}2)$. We then continue the node generation and the node identification process in a breadth-first manner. Obviously, this process is potentially infinite in the sense that from $\mathit{fib}(k)$ we can generate the son node $\mathit{fib}(k{-}i)$ for any $i \geq 0$.

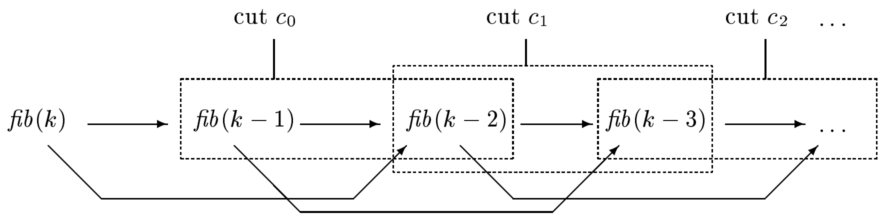


Fig. 1. An initial portion of the m-dag for the *fib* function with the first three cuts of a progressive sequence of cuts.

In Figure 1 we have represented an initial portion of the m-dag for the *fib* function. In constructing this m-dag it is assumed that the argument of every call of the function *fib* is greater than 1, and thus, Equation 3 is used for generating two new nodes from any given node.

Now the general methodology we want to present, requires in its second step, the *search for a suitable regularity* valid in the whole m-dag, and fortunately, as we will see, there is no need for the complete representation of the infinite m-dag.

In our case, a suitable regularity is the existence of a 'progressive sequence of cuts' [38]. Informally, this means that in the m-dag with initial node $\mathit{fib}(k)$ there is a sequence $\langle c_0, c_1, \ldots \rangle$ of sets of nodes with the following properties: i) all sets, also called *cuts*, have equal cardinality, say C, ii) after removing a set of that sequence the resulting m-dag has two disconnected parts (this is why each set of the sequence is called a cut), iii) for any two successive cuts, say c_i and c_{i+1}, we have that: $c_i \neq c_{i+1}$, $\forall n \in c_{i+1}\ \exists m \in c_i$ such that if $n \neq m$ then $m > n$, and $\forall m \in c_i\ \exists n \in c_{i+1}$ such that if $n \neq m$ then $m > n$, where $>$ denotes the transitive closure of the father-son relationship among nodes, iv) there are $1 + C$ functions, say $p_0, p_1, \ldots, p_C$, which all have arity C and are defined in terms of basic functions only, such that: (a) $\mathit{fib}(k)$ can be computed from the C function calls in the cut c_0 using p_0, and (b) $\forall i \geq 0, \forall j$, with $1 \leq j \leq C$, the j-th function call in the cut c_i can be computed from the function calls in the cut c_{i+1} using p_j, and v) for every value of k, with $k \geq 0$, in the sequence of cuts there exists a cut whose function calls, instantiated to that value of k, can be computed using basic functions only, without requiring the computation of the son calls.

A progressive sequence of cuts in the m-dag with initial node $\mathit{fib}(k)$ is: $\sigma = \langle \{\mathit{fib}(k-1), \mathit{fib}(k-2)\},\ \ \{\mathit{fib}(k-2), \mathit{fib}(k-3)\}, \ldots \rangle$.

A different progressive sequence whose cuts have cardinality three, is:
$\langle \{\mathit{fib}(k-1), \mathit{fib}(k-2), \mathit{fib}(k-4)\},\ \ \{\mathit{fib}(k-5), \mathit{fib}(k-6), \mathit{fib}(k-8)\}, \ldots \rangle$.

Then the third step of the general methodology is the *extraction of the new program* from the symbolic computation and the discovered regularity. In our case, given the progressive sequence of cuts σ, we apply the tupling strategy [10, 38] and we introduce the function $t(n) = \langle \mathit{fib}(n+1), \mathit{fib}(n) \rangle$ which for any sequence of values of the variable n, gives us the corresponding sequence of values of the function calls in the cuts of σ. Using the unfold/fold technique [10] we then get the following program:

1. $\mathit{fib}(0) = 1$
2. $\mathit{fib}(1) = 1$
4. $\mathit{fib}(n+2) = u + v$ where $\langle u, v\rangle = t(n)$ for $n \geq 0$
5. $t(0) = \langle 1, 1\rangle$
6. $t(n+1) = \langle u+v, u\rangle$ where $\langle u, v\rangle = t(n)$ for $n \geq 0$

As expected, this program uses only $O(n)$ sums to compute the value of $\mathit{fib}(n)$. The reader should notice that Equations 5 and 6 are obtained in the unfold/fold technique by looking for the explicit recursive definition of the new tuple function $t(n)$. In particular, Equation 6 is derived as follows:

$$\begin{aligned}
t(n{+}1) &= \langle \mathit{fib}(n{+}2), \mathit{fib}(n{+}1)\rangle = \{\text{unfolding}\} = \\
&= \langle \mathit{fib}(n{+}1) + \mathit{fib}(n), \mathit{fib}(n{+}1)\rangle = \{\text{where-abstraction and tupling}\} = \\
&= \langle u{+}v, u\rangle \text{ where } \langle u, v\rangle = \langle \mathit{fib}(n{+}1), \mathit{fib}(n)\rangle = \{\text{folding}\} = \\
&= \langle u{+}v, u\rangle \text{ where } \langle u, v\rangle = t(n)
\end{aligned}$$

The where-abstraction step avoids the double evaluation of $\mathit{fib}(n{+}1)$ while computing $t(n{+}1)$, and the last folding step avoids the double evaluations of the *fib* calls 'at every level of recursion', thus, it makes the efficiency gains of the where-abstraction step computationally significant. This is why in the unfold/fold technique one looks for final folding steps to be made at the end of the derivation. The same occurs, for instance, in the supercompilation technique where one looks for 'self-sufficient models' of the computation [52].

In the following sections we illustrate in some detail the general methodology for program transformation we have seen in action in this preliminary example. We also indicate the way in which various techniques for program transformation proposed in the literature fit into this general methodology. In Section 3 we consider the symbolic computation model called the symbolic trace tree used in compiling control, and we briefly compare it with the models used in other program transformation systems. In Section 4 we illustrate the idea of finding suitable regularities in the symbolic computations, and in particular, we consider the case of partial evaluation in logic programming. In Section 5 we address the problem of extracting new programs from symbolic computations. Since the application of the general methodology is highly nondeterministic and may also lead to infinite constructions, we need some techniques for its control. Those techniques are analyzed in Section 6. In Sections 7 we relate the general methodology to program specialization, deforestation, and finite differencing, and finally, in Section 8 we briefly present some correspondences among various program transformation techniques.

3 Symbolic Computation Models

A method for transforming a given initial program into a new program which behaves efficiently for every input value, is to look for suitable properties which hold for every computation performed by the initial program. These properties can often be discovered by applying the general technique, called *abstract* *interpretation* [13], by which we represent a possibly infinite set of concrete computations, one for every input value, by a single symbolic computation, and then by reasoning on that symbolic computation.

Various models of symbolic computations have been proposed in the literature, and we briefly discuss them at the end of this section. Now we consider in some detail a particular symbolic computation model, called *symbolic trace tree*, which has its relevance in the transformation technique for logic programs called compiling control [7].

The Symbolic Trace Tree for Compiling Control

A logic program can be viewed as the union of some 'logic definitions' (that is, the axioms of a theory) and a 'control strategy' (that is, a theorem prover) [29]. The efficiency of a logic program very often depends on the control strategy. Thus, in order to achieve high performances, the programmer, instead of relying on the evaluation strategy provided by the system, may define his own control strategy. This can be done, for instance, via *modes* or *delay declarations* [33] based on the instantiation patterns of the goals during execution. However, one may avoid the difficulty of dealing with those declarations at run-time by using the *compiling control* technique as we now indicate.

Let S_{left} be the familiar Prolog control strategy, which selects the literals in the goal at hand in a sequential order from left to right. Given a logic program P_1 and an efficient control strategy S_{eff} for P_1, we want to derive a new program P_2 such that, for a given class of goals, P_1 with control strategy S_{eff} and P_2 with control strategy S_{left} have equivalent computational behaviour. According to the general methodology we have presented in Section 2, compiling control works in three steps as follows.

1. Starting from a symbolic input goal, in the first step compiling control generates a *symbolic trace tree* using the control strategy S_{eff}. The symbolic trace tree represents the class of concrete computations, each of which corresponds to a concrete goal in the class of goals represented by the symbolic input goal.
2. We then look for a *finite* description of the symbolic trace tree which is potentially infinite. This is done by identifying *similar* nodes and thus, generating a finite graph, possibly cyclic, called *symbolic trace graph.* The notion of similarity may vary according to the particular instances of the compiling control technique one uses.
3. In the final third step a new program P_2 is extracted from the symbolic trace graph. By construction, the behaviour of P_2 with the control strategy S_{left} is equivalent to that of P_1 with control strategy S_{eff}.
   This equivalence establishes the correctness of the transformation and it is based on the relationship between the concrete and the symbolic computations which is formalized, as we will see in the example below, by using the abstract interpretation technique.

Ideas related to compiling control have also been investigated in the area of functional programming within the so called *filter promotion* strategy [4, 14], whereby function evaluations can be anticipated for avoiding unnecessary computations and improving program behaviour.

In the following example we will see in action the compiling control technique. The final program can also be derived by using unfold/fold program transformations as shown in [50].

Example 1. [*Common Subsequences*] Let us consider the following logic program *Csub*, which generates all common subsequences X of not necessarily consecutive elements of two sequences Y and Z. Sequences are represented as lists.

1. $\mathit{csub}(X,Y,Z) \leftarrow \mathit{subseq}(X,Y), \mathit{subseq}(X,Z)$
2. $\mathit{subseq}([\,],X) \leftarrow$
3. $\mathit{subseq}([A|X],[A|Y]) \leftarrow \mathit{subseq}(X,Y)$
4. $\mathit{subseq}(X,[B|Y]) \leftarrow \mathit{subseq}(X,Y)$

Let us consider the set I of input goals of the form $\mathit{csub}(x,y,z)$, where x is a free variable and y and z are ground lists. For these goals the control strategy S_{left} is, in general, inefficient because it first evaluates $\mathit{subseq}(x,y)$ and generates a binding, say $\bar{x}$, for x and then it tests whether or not $\mathit{subseq}(\bar{x},z)$ holds.

The following *producer-consumer* coroutining strategy, called S_{pc}, allows for a more efficient execution of the above program. This strategy assumes that an atomic goal A is said to be a *consumer* (of bindings) iff all its arguments are instances of the arguments of every clause head which is unifiable with A itself, and otherwise the atom A is said to be a *producer* (of bindings). The strategy S_{pc} can be defined as follows: in the goal at hand S_{pc} chooses for execution the leftmost consumer, if any, and otherwise it chooses the leftmost producer.

In order to represent a set of concrete goals as a single symbolic goal, we consider the set G of all ground terms and the set F of all free variables. These two sets, together with the empty set of terms and the set of all (ground and non-ground) terms, form the domain of an abstract interpretation which is a lattice. (The reader unfamiliar with abstract interpretations in logic programming may refer to [2, 6, 32].)

A finite portion of the symbolic trace tree for a goal in I, generated by using the program *Csub* and the control strategy S_{pc}, is depicted in Figure 2 (where for the time being, the upgoing arrows are to be ignored). The root is labeled by the symbolic input goal $\mathit{csub}(X^F,Y^G,Z^G)$, meaning that in every concrete computation the input goal is of the form $\mathit{csub}(X,Y,Z)$ where X is a free variable and Y and Z are bound to ground terms. In the goals labeling the non-root nodes of that tree, a variable with superscript F means that in every concrete computation that variable is bound to a (possibly different) free variable, whereas a variable with superscript G means that in every concrete computation that variable is bound to a ground term.

In the node M we have unfolded the atom $\mathit{subseq}(X^F,Y^G)$ because it unifies either with clause 2 (if $Y^G = [\,]$) or with clauses 3 and 4 (if Y^G is a non-empty ground list). In both cases $\mathit{subseq}(X^F,Y^G)$ is a producer, and for the same reasons, also $subseq(X^F, Z^G)$ is a producer. In the node N we have unfolded the atom $subseq([A^F|X1^F], Z^G)$ because it is a consumer and $subseq(X1^F, Y1^G)$ is a producer.

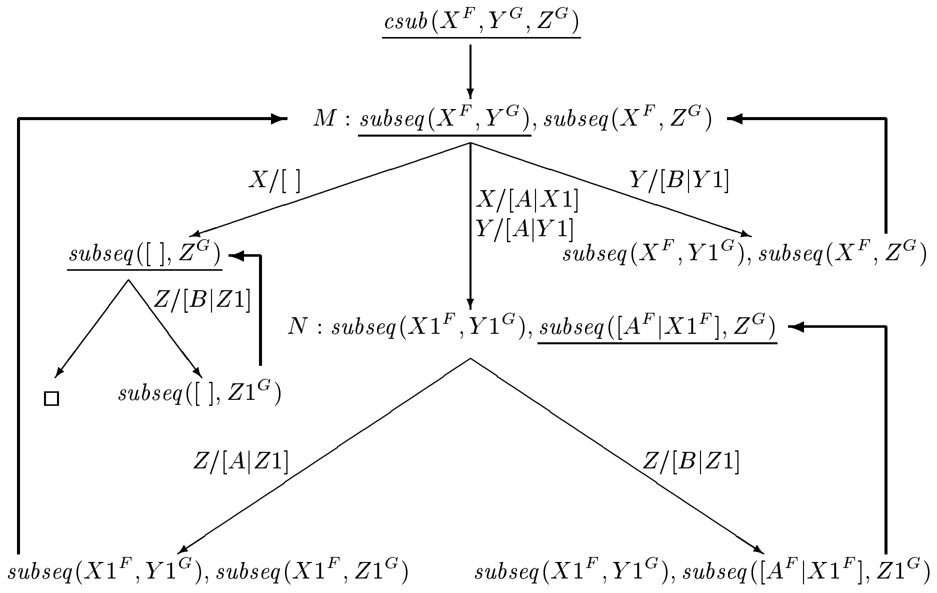

$\underline{csub(X^F, Y^G, Z^G)}$
$M : \underline{subseq(X^F, Y^G)}, subseq(X^F, Z^G)$
$X/[\,]$
$X/[A|X1]$
$Y/[A|Y1]$
$Y/[B|Y1]$
$\underline{subseq([\,], Z^G)}$
$subseq(X^F, Y1^G), subseq(X^F, Z^G)$
$Z/[B|Z1]$
$N : subseq(X1^F, Y1^G), \underline{subseq([A^F|X1^F], Z^G)}$
$\square$
$subseq([\,], Z1^G)$
$Z/[A|Z1]$
$Z/[B|Z1]$
$subseq(X1^F, Y1^G), subseq(X1^F, Z1^G)$
$subseq(X1^F, Y1^G), subseq([A^F|X1^F], Z1^G)$

Fig. 2. An initial portion of the symbolic trace tree for *Csub*. The atoms selected for unfolding by the strategy S_{pc} are underlined. Upgoing arrows relate symbolic goals which are variants of each other. These arrows do *not* denote arcs of the tree.

The goal labeling any non-root node of the symbolic trace tree is obtained from the goal of the corresponding father node according to the following unfolding process: i) we select an atom of the goal in the father node following the strategy S_{pc}, ii) we unify the selected atom with the heads of all clauses in *Csub*, iii) we replace the selected atom by the bodies of the unifying clauses, whereby getting the son nodes, and iv) we apply to the son nodes the bindings computed by unification. In the symbolic trace tree the arc from a father node to a son node is labeled by the bindings for the variables of the father node that are computed during the corresponding unfolding step by the unification process.

The variable superscripts in any son node are obtained from the superscripts in the corresponding father node by taking into account that: i) the unification of a ground term with a term containing variables binds all variables to ground terms, and ii) the unification of a variable with a term containing variables does not bind any variable in that term. We leave to the reader the task of formalizing the process of computing the variable superscripts. This can be done by using the notion of *abstract unification*, that is, unification among terms in the domain of the abstract interpretation [2, 6, 32].

Now, as in the second step of the general methodology, compiling control searches for regularities in the symbolic trace tree with the aim of deriving a finite representation of that tree. In our case this finite representation can be obtained by identifying goals which are variants of each other and have the same superscripts. By doing so we get the finite cyclic graph, called *symbolic trace graph*, depicted in Figure 2 where nodes related by upgoing arrows are to be identified.

The theory of abstract interpretation can be used for proving various correctness properties of the symbolic trace graph and in particular, the fact that it indeed represents the set of all concrete computations generated by the given set of input goals, in the sense that every concrete computation follows a sequence of arcs in that graph and at each computation step the concrete goals are instances of the symbolic goals in the corresponding nodes and they agree with the superscripts.

As we will discuss in the next section, this finite representation property has a fundamental importance and it allows us to perform the third step of the general methodology, that is, the derivation of a new program from the initial one.

Finiteness of the symbolic trace graphs is related to analogous properties which are required in other transformation techniques, such as *self-sufficiency* of the graphs of states and transitions in supercompilation [52], *foldability* of the unfolding trees in unfold/fold program transformation [43], and *closedness* in partial deduction [31].

The extraction of the derived program is performed as we now indicate (see also Figure 3).

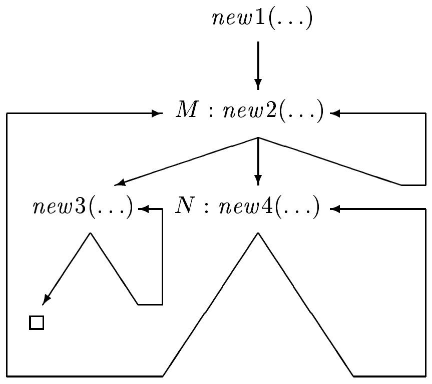


Fig. 3. The symbolic trace graph derived from the symbolic trace tree of Figure 2 after the introduction of the new predicate names: $new1$, $new2$, $new3$, and $new4$.

We introduce new predicate names, say $new1, new2, \ldots$, one for each set of non-empty variant goals, and for each arc $U \xrightarrow{\theta} V$ of the symbolic trace tree, with $V \neq \Box$, we introduce the new clause $new\text{h}(U)\theta \leftarrow new\text{k}(V)$, where the predicate names occurring in U and V are to be considered as function names, because they now occur in argument positions. For the arc $subseq([\,], Z) \to \Box$ we introduce the new clause $new3(subseq([\,], Z)) \leftarrow$.

Thus, we get the following new program:

$$
\begin{array}{l}
new1(csub(X,Y,Z)) \leftarrow new2(subseq(X,Y), subseq(X,Z)) \\
new2(subseq([\,],Y), subseq([\,],Z)) \leftarrow new3(subseq([\,],Z)) \\
new2(subseq([A|X1],[A|Y1]), subseq([A|X1],Z)) \leftarrow \\
\qquad\qquad\qquad new4(subseq(X1,Y1), subseq([A|X1],Z)) \\
new2(subseq(X,[B|Y1]), subseq(X,Z)) \leftarrow \\
\qquad\qquad\qquad new2(subseq(X,Y1), subseq(X,Z)) \\
new3(subseq([\,],Z)) \leftarrow \\
new3(subseq([\,],[B|Z1])) \leftarrow new3(subseq([\,],Z1)) \\
new4(subseq(X1,Y1), subseq([A|X1],[A|Z1])) \leftarrow \\
\qquad\qquad\qquad new2(subseq(X1,Y1), subseq(X1,Z1)) \\
new4(subseq(X1,Y1), subseq([A|X1],[B|Z1])) \leftarrow \\
\qquad\qquad\qquad new4(subseq(X1,Y1), subseq([A|X1],Z1))
\end{array}
$$

For goals of the form $new1(csub(X,Y,Z))$, this program computes the same answers as the ones computed by $Csub$ for goals of the form $csub(X,Y,Z)$ where X is a free variable and Y and Z and ground terms. The derived program, however, is more efficient than the initial one because it is more deterministic.

In order to avoid the presence of nested terms, intermediate predicates, and subsumed clauses, we then perform some final simple transformations which are similar to 'post-unfolding' in the supercompilation technique. Thus, we get the following program [50]:

$$
\begin{array}{l}
new5([\,],Y,Z) \leftarrow \\
new5([A|X],[A|Y],Z) \leftarrow new6(A,X,Y,Z) \\
new5(X,[B|Y],Z) \leftarrow new5(X,Y,Z) \\
new6(A,X,Y,[A|Z]) \leftarrow new5(X,Y,Z) \\
new6(A,X,Y,[B|Z]) \leftarrow new6(A,X,Y,Z)
\end{array}
$$

where the predicates $new5$ and $new6$ correspond in Figure 2 (and 3) to node M and N, respectively. $\Box$

Other Symbolic Computation Models

Now we would like to consider some other symbolic computation models which have been proposed in the literature both for functional and logic languages. They differ for the information which is recorded during the symbolic computation steps. However, all of them use a basic operation similar to unfolding, which may be viewed as an abstraction of a computation step. The specific form of this basic operation varies in accordance with the language and the semantics considered.

Burstall and Darlington [10] have the *execution tree* model which is used to discover the new function definitions, the so called *eureka definitions*, to be introduced during the derivation of new programs via *folding/unfolding* transformations. The execution tree may be viewed as an abstraction of the concrete computation and consists of a tree of recursive calls constructed by unfolding a symbolic input term. The *m-dags* of recursive calls [3, 38], which have been presented in our preliminary example, are further developments of this approach. A symbolic computation model based on unfolding, the so called *unfolding tree*, that is, a tree of clauses obtained by unfolding, has been proposed for logic programming in [43, 41].

In Turchin's supercompilation technique [52], the symbolic computation process is performed by *driving*, which is analogous to unfolding. The driving process generates a tree of configurations, or a graph if we identify nodes with *similar* configurations. This graph is called *graph of states and transitions*. A similar model is the *partial process tree* used in the *positive supercompilation* technique [47, 48]. There is, however, a difference between unfolding à la Burstall and Darlington and driving à la Turchin: by unfolding we replace an expression which matches the left-hand side of an equation by the corresponding instance of the right-hand side, whereas by driving a sort of unification process, rather than matching, takes place. This makes driving very similar to the unfolding mechanism we have seen in action in the compiling control example above. A formal correspondence between driving and unfolding in logic programming can be found in [25].

More similarities between supercompilation and other techniques used for transforming logic programs are based on the idea of performing symbolic computations by *meta-programs* or, in Turchin's terminology, *metasystem transitions* [24, 54]. For instance, the transformation technique presented by Gallagher in [20] works by specializing a *meta-interpreter*, that is, a logic program which works as an interpreter for logic programs, w.r.t. a particular input program. Also the symbolic trace tree for compiling control may be generated using a meta-interpreter.

More complex operations may be performed during symbolic computation. For instance, in supercompilation one is allowed to use any 'clever trick' [52, page 293], in GPC-trees [19] one may use theorem provers to partially evaluate conditionals, and when constructing unfolding trees of logic programs one may perform, together with unfolding and folding steps, also goal replacement steps [41]. By these goal replacement steps we replace old goals by new equivalent goals using lemmas whose proofs are done off-line.

A special model of symbolic computation is the *SLDNF-tree* which is the basis for the partial evaluation technique in logic programming [31]. In this model the symbolic computation coincides with the concrete one (which can also be represented as an SLDNF-tree), because in logic programming one is allowed to run programs with input goals which contain free variables. Further refinements of partial evaluation, such as the techniques based on *characteristic trees* [22, 30], use notions which are abstractions of SLDNF-trees.

4 Searching for Regularities in Symbolic Computations

In this section we consider the problem of searching for regularities in a symbolic computation model of the program at hand. These regularities may be used for extracting a new program.

It is hard to devise a general notion of regularity which ensures that the derived programs are in all cases more efficient than the initial ones. Thus, different notions of regularity have been considered in the various program transformation techniques. Those notions, however, are not unrelated, and indeed most of them refer to *similarity* relations which hold between nodes of symbolic computation models. In particular, let us consider again the compiling control example of the previous section. In that example we have seen that an efficient program which embodies an enhanced control strategy, can be extracted from the symbolic trace tree when each leaf goal is a variant of an ancestor goal and the corresponding renaming substitution preserves the instantiation of the variables (that is, the superscripts G and F). This correspondence between nodes of the symbolic trace tree is the similarity used in compiling control.

In other symbolic computation models one may find other similarity relations (not necessarily symmetric) which formalize the fact that a configuration (or a set of configurations) of the symbolic computation can be expressed in terms of a previously generated configuration (or set of configurations). For instance, in the case of the m-dag model described in the Fibonacci example, a cut may be considered to be similar to the next cut in a progressive sequence [38], because we can get the function calls in a cut from those of the next cut by substituting an expression for a variable (in our case, $k-1$ for k).

Since we should be able to derive a new program from the symbolic computation of a given initial program, it is important that we find a *finite* representation of the potentially infinite symbolic computation, because as we will see in the next section, the structure of the derived program is closely related to that of the symbolic computation. This explains why several notions of regularity require that one should find a finite set of configurations such that every configuration in the symbolic computation is similar to a configuration in that set.

Now we look at the partial evaluation technique in logic programming and we indicate the particular notions of similarity and regularity which are used there. We will then mention how these concepts are used in other transformation techniques.

Regularities in Partial Evaluation of Logic Programs

Partial evaluation is a well-known program transformation technique which allows us to derive a new program from an old one when part of the input data is known before evaluation. The reader may refer to [12, 21, 26] for introductions and surveys on this topic.

In the case of logic programming, where partial evaluation is also called *partial deduction*, it is usually assumed that we are given an initial program P and a set A of possibly non-ground atoms, and by partial deduction of P w.r.t.

A, we want to derive a new program P' such that P and P' compute the same answers for every input goal which is an instance of an atom in A.

One of the most popular techniques for partial deduction has been formalized by Lloyd and Shepherdson [31]. In that technique the program P' is obtained by collecting together the clauses, called *resultants*, which are constructed as follows: for each element A_i of A, i) we first construct a finite portion, containing more than one node, of an SLDNF-tree, say T_i, for the program P and the atom A_i, then ii) we consider the non-failing branches of T_i and the goals at their leaves, say $B_{i1}, \ldots, B_{ir}$, and the computed substitutions along these branches, say $\theta_{i1}, \ldots, \theta_{ir}$, and finally, iii) we construct the clauses: $A_i\theta_{i1} \leftarrow B_{i1}, \ldots, A_i\theta_{ir} \leftarrow B_{ir}$.

The SLDNF-trees constructed for partial deduction can be viewed as symbolic computations starting from the atoms in A and representing all SLDNF-trees starting from atoms which are instances of the atoms in A.

If we now assume that: i) every atom in P' is an instance of an atom in A, that is, P' is A-*closed*, and ii) no two atoms in A have a common instance, that is, A is an *independent* set of atoms, then P' is a *correct* partial deduction of P w.r.t. A, in the sense that for every input goal G which is an instance of an atom in A, we have that $P \cup \{\leftarrow G\}$ has a computed answer substitution θ iff $P' \cup \{\leftarrow G\}$ has the computed answer substitution θ, and $P \cup \{\leftarrow G\}$ finitely fails iff $P' \cup \{\leftarrow G\}$ finitely fails [31]. The following example shows that partial deduction can be viewed as a particular case of our general program transformation methodology made out of three steps.

Example 2. [*Partial Deduction of a Parser*] Let us consider the following *Parse* program, adapted from [49, page 381], for parsing words of context free languages.

$$
\begin{array}{l}
parse(Grammar, [Symb], [Symb|X]\backslash X) \leftarrow terminal(Symb)\\
parse(Grammar, [Symb], Word) \leftarrow nonterminal(Symb),\\
\qquad\qquad member(Symb \rightarrow Symbs, Grammar),\\
\qquad\qquad parse(Grammar, Symbs, Word)\\
parse(Grammar, [Symb1, Symb2|Symbs], WordX\backslash X) \leftarrow\\
\qquad\qquad parse(Grammar, [Symb1], WordX\backslash Y),\\
\qquad\qquad parse(Grammar, [Symb2|Symbs], Y\backslash X)\\
terminal(0) \leftarrow\\
terminal(1) \leftarrow\\
nonterminal(s) \leftarrow\\
nonterminal(u) \leftarrow
\end{array}
$$

The first argument of *parse* is a grammar represented as a list of productions of the form $Symb \rightarrow Symbs$, where $Symb$ is a nonterminal symbol and $Symbs$ is a sequence of terminal or nonterminal symbols. The second argument of *parse* is a list representing the sentential form at hand. The third argument is the word W to parse which is represented as a difference-list, that is, $X\backslash Y$ is the difference list representing W iff W concatenated with Y is X. This representation allows for an efficient word decomposition, which is needed in the third clause of *parse*.

Suppose that we want to specialize our parser w.r.t. the grammar

$\{s \to 0\ u, \quad u \to 1, \quad u \to 0\ u\ u, \quad u \to 1\ s\}$

where s is the start symbol. In other words, we want to partially evaluate *Parse* w.r.t. the input goal $parse(\Gamma, [s], X\backslash[\,])$ where Γ is the term $[s \to [0,u], u \to [1], u \to [0,u,u], u \to [1,s]]$ representing the given grammar. To this aim, we consider the following independent set A of two atoms:

$A = \{parse(\Gamma, [s], X\backslash Y),\ parse(\Gamma, [u], X\backslash Y)\}$

which will allow us to partially evaluate the given program w.r.t. the input goal $parse(\Gamma, [s], X\backslash[\,])$ because this goal is an instance of the atom $parse(\Gamma, [s], X\backslash Y)$ in that set [31].

We then construct the two finite initial portions T_1 and T_2 of SLDNF-trees (in this case no negation as failure steps are needed, because the program is positive) depicted in Figure 4. In this figure: i) an arc stands for one or more SLDNF-resolution steps, ii) underlined atoms are the ones which are unfolded, iii) when not indicated the substitution corresponding to a successful arc is the identity substitution, iv) $\times$ denotes failure, v) $\Box$ denotes success, and vi) upgoing arrows relate leaf goals to root goals of which they are instances (these arrows are not arcs of the SLDNF-trees).

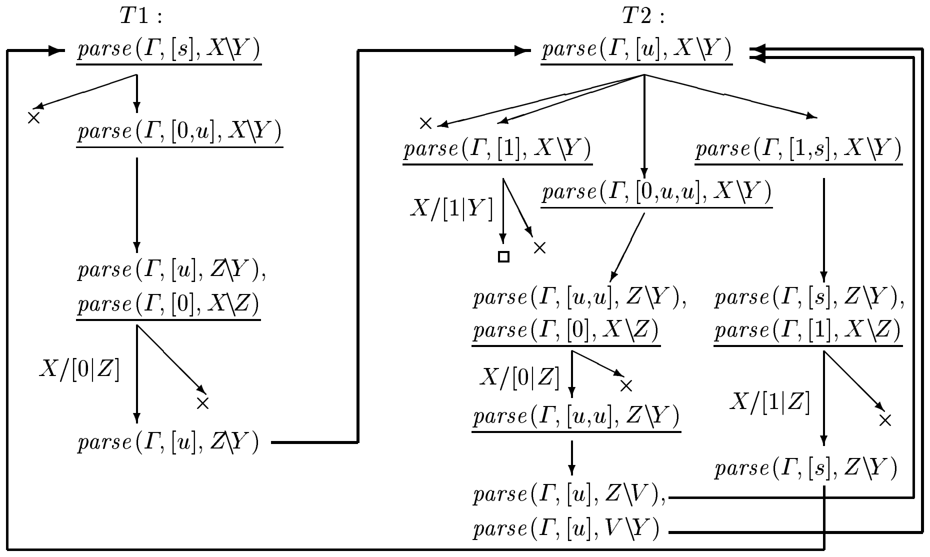


Fig. 4. Two finite portions T_1 and T_2 of SLDNF-trees for the *Parse* program.

The resultants of the non-failing branches of these two SLDNF-trees are the following clauses:

$parse(\Gamma, [s], [0|Z]\backslash Y) \leftarrow parse(\Gamma, [u], Z\backslash Y)$
$parse(\Gamma, [u], [1|Y]\backslash Y) \leftarrow$

$parse(\Gamma, [u], [0|Z]\backslash Y) \leftarrow parse(\Gamma, [u], Z\backslash V), parse(\Gamma, [u], V\backslash Y)$
$parse(\Gamma, [u], [1|Z]\backslash Y) \leftarrow parse(\Gamma, [s], Z\backslash Y)$

These resultants form an A-closed set of clauses and therefore they constitute a correct partial deduction of *Parse* w.r.t. A.

Similarly to the compiling control example, the above program may be further improved by introducing, using the *renaming* transformation [21], new predicate names corresponding to the different instances of the predicate *parse*. By doing so (and also by renaming some variables), we get the following final program *PdParse*:

$pd\_parse(X) \leftarrow parse\_s(X, [\,])$
$parse\_s([0|X], Y) \leftarrow parse\_u(X, Y)$
$parse\_u([1|X], X) \leftarrow$
$parse\_u([0|X], Y) \leftarrow parse\_u(X, Z), parse\_u(Z, Y)$
$parse\_u([1|X], Y) \leftarrow parse\_s(X, Y)$

The correctness results for partial deduction can easily be extended to the case where we consider the above renaming transformation. In this example we have that, for every term w, $Parse \cup \{\leftarrow parse(\Gamma, [s], w\backslash[\,])\}$ has a computed answer substitution θ iff $PdParse \cup \{\leftarrow pd\_parse(w)\}$ has the computed answer substitution θ, and $Parse \cup \{\leftarrow parse(\Gamma, [s], w\backslash[\,])\}$ finitely fails iff $PdParse \cup \{\leftarrow pd\_parse(w)\}$ finitely fails. □

One may notice that the A-closedness property of the program derived by partial deduction from a program P w.r.t. A, is equivalent to the following property of the set of SLDNF-trees constructed for the partial deduction of P w.r.t. A: each atom occurring in a non-failing leaf of an SLDNF-tree in this set of trees is an instance of an atom occurring in a root (not necessarily within the same tree). If a set of SLDNF-trees enjoys this property we will say that it is *closed* (see Figure 4 for an example where this property holds).

The notion of a closed set of SLDNF-trees nicely illustrates the idea of regularity of a symbolic computation. Indeed, that notion is based on the similarity relation whereby a leaf node is similar to a set of root nodes iff every atom in the goal of that leaf is an instance of the atom in a root of the given set. We also have that a closed SLDNF-tree T, that is, a closed set of SLDNF-trees with one tree only, represents an infinite SLDNF-tree where all goals may be expressed in terms (more precisely, are conjunctions of instances) of the finite set of goals occurring in T. Moreover, given a closed set of SLDNF-trees we may derive a program which is equivalent to, and hopefully more efficient than the initial program, by extracting clauses from their non-failing root-to-leaf paths.

Other Notions of Regularity

We have mentioned at the beginning of this section the notions of regularity used in compiling control and in some unfold/fold techniques for functional programs. Now we will briefly discuss some other forms of regularities in symbolic computation models.

One of the earliest transformation techniques which uses concepts analogous to 'similarity' and 'regularity', is Turchin's supercompilation. As already mentioned in Section 1, the symbolic computation model for supercompilation is the directed graph of states and transitions constructed by driving and generalization steps (In Section 6 we will give more details on the generalization operation). In this directed graph a configuration C_j is similar to a previously generated one C_i iff C_j is a *specialization* of C_i, that is, the set of concrete computation states represented by C_j is a subset of that of C_i. The notion of regularity corresponds to that of *self-sufficiency*: a finite graph of states and transitions is said to be self-sufficient when every configuration is either *passive* (that is, an expression made out of basic operators) or similar to a previously generated one.

Related concepts of similarity and regularity are also present in various versions of the supercompilation technique, such as *positive supercompilation* [48].

Also in the unfold/fold technique for the transformation of logic programs, we encounter a similarity notion and a regularity notion. They are related to the construction of unfolding trees [43] which are used for guiding the application of the unfold/fold rules. The similarity notion is the *foldability of a clause* and we say that a clause is foldable when its body (except for some basic predicates) is an instance of the body of an ancestor clause in the tree. The regularity notion is the *foldability of the unfolding tree* and we say that an unfolding tree is foldable when it has a finite upper portion whose leaves are foldable clauses or clauses whose bodies either are made out of basic predicates or contain failures. The reader will realize the very close relationship between the notions of foldable unfolding trees and closed SLDNF-trees we have presented above.

In the generalized partial computation technique the similarity notion between nodes in a GPC-tree is determined by the absence of the so called P-redexes [19]. In particular, in a GPC-tree a node N is similar to an ancestor node M if i) $\lambda x.f(x)$ is the function computed at node M with domain dom_M, ii) $\lambda x.A[f(B[x])]$ is the function computed at node N where $A[\ldots]$ and $B[\ldots]$ are suitable contexts and $B[x]$ ranges over dom_N, and iii) $dom_M \subseteq dom_N$. The regularity notion of a GPC-tree is, as usual, based on the fact that every leaf node is either a basic value or similar to an ancestor node. We cannot go into more details here. However, we want to stress that, in sharp contrast to supercompilation, in generalized partial computation one performs an unfolding step when a node represents a subset of the set of the concrete computation states represented by a previously generated node. The underlying assumption is that with more information on the input data one may get more specialized and hopefully, more efficient programs. We will return on this issue of specialization versus generalization in Section 6.

5 Program Extraction

In this section we consider the third step of the general methodology, that is, the process of extracting a new program from the symbolic computation of the given initial program and also the suitable regularities which have been discovered.

We have already remarked that it is important that the regularities are 'suitable', that is, they indeed allow for program extraction. We are not interested here in the formalization of this suitability notion. It will be enough to consider the particular case, which is the most frequent in practice, where the symbolic computation is described by means of a directed graph whose arcs correspond to concrete computation steps. There are basically two approaches to program extraction in this case: either the *direct extraction* or the *extraction via transformation rules*.

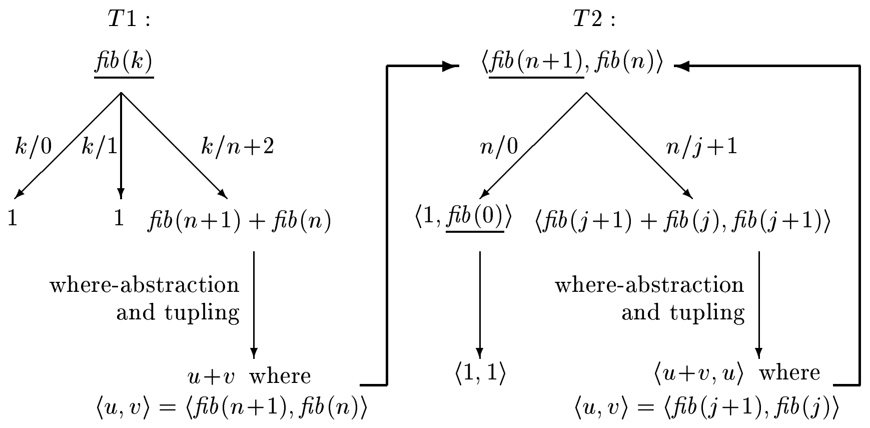
An example of the first approach can be taken from partial deduction, where there is a simple way of deriving the clauses of the final program directly from the closed set of SLDNF-trees which have been constructed by applying, for instance, the procedure described in [21]. Indeed, every path from the root to a non-failing leaf in those trees, generates a clause of the program to be extracted. Also in generalized partial computation we directly extract programs from the corresponding GPC-trees by looking at their paths, but we may also allow for the use of some recursion removal techniques [19].

An example of program extraction via transformation rules may be given using the transformation of the Fibonacci program presented in Section 2. We start from the known regularity, that is, the existence of a progressive sequence of cuts, and we perform the extraction by exploiting the properties of that sequence as follows: i) we first introduce by the definition rule the new function $t(n)$ which tuples together the function calls in a generic cut, ii) we apply the unfolding rule and the where-abstraction rule to express the initial function call in terms of the calls in the first cut whereby extracting Equation 4, that is, $\mathit{fib}(n+2) = u+v$ where $\langle u,v\rangle = t(n)$, iii) by applying the unfolding rule we compute the value of the function calls in the cut for which there is no need to compute the calls of their son nodes whereby extracting Equation 5, that is, $t(0) = \langle 1,1\rangle$, and finally, iv) we apply the unfolding, where-abstraction, and folding rules to compute the values of the function calls in a cut from those in the next cut whereby extracting Equation 6, that is, $t(n+1) = \langle u+v, u\rangle$ where $\langle u,v\rangle = t(n)$.

This program extraction shows that for the *fib* function the existence of a progressive sequence of cuts is a suitable regularity, and indeed the various properties of that sequence suggest the actions to be performed during the extraction itself.

For the extraction of the new program of the *fib* function we can also use the direct approach, but it is necessary to construct a symbolic computation model which is more informative than the one based on the m-dags used in Section 2. This more informative model is based on the construction of a set of trees which can be obtained by an extension of the positive supercompilation technique described in [47]. The extension is motivated by the fact that in positive supercompilation it is not possible to directly exploit the interactions among different function calls because they belong to different branches of the process trees. We will not give here the formal rules for the construction of this more informative model of computation. It will be enough to say that we follow closely the positive supercompilation and partial deduction techniques. There are, however, some differences. In particular, the differences w.r.t. positive supercompilation include the rules that: i) when folding can be performed we do not expand the process tree and we initialize a new tree, instead, and ii) when constructing trees we perform together with unfolding and lemma application steps, also where-abstraction steps and tupling steps. Tupling consists in the introduction of new functions defined in terms of tuples of some old functions. These tuples of functions are those which allow us to take advantage of the interaction among different function calls. Differences w.r.t. partial deduction include: i) a new notion of the similarity relation among nodes: a node N_1 with label $expr_1$ is similar to a node N_2 with label $C[expr_2]$ iff the expressions $expr_1$ and $expr_2$ are variants of each other, and $C[\ldots]$ is a context made out of basic functions only, and ii) a new notion of closedness: a set of trees is said to be closed iff every leaf of every tree has a similar root (according to the new notion we have now introduced).

A possible criterium for terminating the construction of a tree is as follows: a node of a tree is a leaf if either i) it cannot be subject to any unfolding or ii) it is similar to a root or iii) it has been produced by a tupling step. In this last case we initialize a new tree whose root is the tuple which has been introduced by the tupling step.



$T1:$
$\underline{fib(k)}$
$k/0$
$k/1$
$k/n+2$
1
1
$fib(n+1)+fib(n)$
where-abstraction and tupling
$u+v$ where $\langle u,v\rangle = \langle fib(n+1), fib(n)\rangle$
$T2:$
$\underline{\langle fib(n+1), fib(n)\rangle}$
$n/0$
$n/j+1$
$\langle 1, \underline{fib(0)}\rangle$
$\langle fib(j+1)+fib(j), fib(j+1)\rangle$
$\langle 1, 1\rangle$
where-abstraction and tupling
$\langle u+v, u\rangle$ where $\langle u,v\rangle = \langle fib(j+1), fib(j)\rangle$

Fig. 5. Two trees which represent a symbolic computation of the *fib* function. Underlined expressions are the unfolded ones. Leaves are related to their similar roots by upgoing arrows. Those arcs are *not* arcs of the trees.

In Figure 5 we have depicted a set of trees which represent the symbolic computation of the *fib* function from which we can get the new program for *fib* by direct extraction. Given the trees of Figure 5 the extraction of the new program is performed as follows. Analogously to what we have seen in the previous section for partial deduction, we first give a new name to the new roots we have introduced ($\langle fib(n+1), fib(n)\rangle$ only in our case). This operation in the unfold/fold technique corresponds to the definition of 'eureka predicates' [10], and in the partial deduction technique corresponds to renaming [21]. We then perform some folding steps corresponding to the nodes which have similar roots, and we finally extract an equation for each root-to-leaf path in the trees obtained after folding, by taking into account also the substitutions along the paths.

6 Control Issues

The various steps of the general methodology for program transformation and in particular, the symbolic computation process must be controlled in some way if one wants to derive very efficient programs. Now we briefly consider these control issues which can be classified into two different categories: *local* and *global* control issues.

6.1 Local Control Issues

When the symbolic computation is performed via unfolding steps (or driving), we may get into *non-deterministic* situations, whereby the symbolic computation steps may allow for more than one successor expression (or configuration). For instance, during the symbolic execution of a logic program often we may choose in more than one way the atom to unfold, and analogously, during the symbolic execution of a functional program often there is a choice of the expression to be evaluated in the following execution step. Different choices may drastically affect the following step of the general methodology, that is, the search for suitable regularities. These choices may be subject to constraints, like for instance, the fact that the symbolic computation should preserve the semantics of the concrete computations it represents. For example, in functional programs if we consider the call-by-value semantics, the innermost function calls should be evaluated before the outermost ones.

Many techniques for controlling unfolding and driving have been proposed in the various models of symbolic computations (see for instance, [11, 43, 55] and the preceding compiling control example). In particular, several authors have studied the problem of when to stop unfolding, and for instance, one may decide to do so when unfolding is no longer deterministic [21] or when the expression in the node at hand can be 'embedded' in the expression in one of its ancestors [47]. A general technique for ensuring the termination of the unfolding process is described in [8].

During the construction of a symbolic computation model, in order to derive programs with high performances it is often important, in practice, to perform *lemma application* steps, that is, to substitute subexpressions by equivalent new subexpressions. The reader familiar with program derivation techniques knows that these lemma applications, also called *law applications* in functional programming or *goal replacements* in logic programming, may allow for a great improvement of program performances which is otherwise impossible (see [26, 56] for some upper bounds on the program speedups which can be obtained without the use of lemmas). Typical lemmas one wishes to apply are: associativity of concatenation, existence of a neutral element for *plus* and *times*, etc. Actually, these lemmas should preserve equivalence of the whole expressions where the substitutions take place, and therefore, they should determine congruences, not simply equivalences.

The control issue related to these lemma applications concerns the problem that while generating the symbolic computation model, one has to decide which lemma should be applied and where it should be applied. This is an important issue which does not have a general solution, because unfortunately, there is no theory by which in all cases we may guide the search and the application of lemmas so that a suitable regularity will eventually be discovered.

The ability to perform lemma application steps makes program transformation very closely related to theorem proving, and indeed some people have looked at techniques which allow for an easy integration of the two areas, by for instance, making derivations and proofs in the same transformational style. Among other techniques we want to mention the *unfold/fold proof method*, which can be used both for program transformation à la Burstall-Darlington and for equivalence proofs. This unfold/fold method can be traced back to Scott (see [10]) and Kott [28] in the case of functional programs, and in the case of logic programs it has been recently presented in [45]. Also Turchin in [52, page 293] explicitly refers to the interaction of theorem proving and program transformation. He advocates the use of theorem provers for the discovery of 'clever properties' when deriving new configurations from old configurations, and he shows how one can, in principle, use supercompilation for proving theorems (see also [51]). In [19] the interaction between theorem proving and program derivation is used for generalized partial computation which is an enhanced partial evaluation technique.

Local control issues also include the decision of when and where to apply the composition strategy and the tupling strategy during program transformation (although for some aspects one may also consider that these strategies do refer to global control issues). We consider the composition strategy in the next section when presenting the deforestation technique, while we have already seen the tupling strategy in action in the *fib* example in the previous section. The composition strategy may generate efficient programs because it may avoid the construction of unnecessary intermediate data structures, while the tupling strategy may avoid repeated subcomputations because it groups together function calls which share the same variables.

6.2 Global Control Issues

In this category of control issues we consider those which are related to the problem of generating a *finite* symbolic computation model with suitable regularities.

We first consider the *generalization* issue, whereby instead of generating the computation model for the function (or predicate) at hand, we generate the model for a generalization of that function (or predicate). The advantage of this technique is that, for the notions of similarity one uses in practice, the generalized function generates configurations which are similar to already constructed ones more often than the non generalized function. Thus, it may be the case that functions with infinite computation models have generalizations with finite models. This situation is analogous to the one often encountered in theorem proving, whereby if a given lemma cannot be proved by induction, one may look for a suitable generalization with the hope of successfully performing an inductive proof of the generalized lemma. Indeed, the new variables introduced by generalization may allow new matches among expressions, and thus, one may perform the proof of the generalized lemma by applying the stronger inductive hypothesis. An application of the generalization technique occurred, in particular, in the partial deduction of the *Parse* program (see Section 4), where the atom $parse(\Gamma,[s],X\backslash[\,])$ has been generalized to $parse(\Gamma,[s],X\backslash Y)$.

In program transformation, generalization is often motivated by the need for folding [14], that is, the need of considering an expression as an instance of another. Thus, generalization is realized by promoting some subexpressions to variables, and usually one considers the most specific common generalization of the two expressions at hand. Sometimes, however, one has to allow for *higher order generalizations* (also called *lambda abstractions* [42]), by which an expression, say $C[e]$, is replaced by the function application $(\lambda x.C[x])e$ where the subexpression e has been promoted to the bound variable x. Here is a simple example of program derivation using higher order generalization.

Example 3. [*Palindrome*] The following program tests whether or not a given list l is a palindrome:

1. $palin(l) = eqlist(l, rev(l))$
2. $eqlist([\,], l) = null(l)$
3. $eqlist(a{:}l_1, l) \;=\; (a = hd(l))\; and\; eqlist(l_1, tl(l))$
4. $rev([\,]) = [\,]$
5. $rev(a{:}l) = rev(l) :: [a]$

where : and :: stand for the operators *cons* and *append*, respectively, $null(l) = true$ iff $l = [\,]$, and *hd* and *tl* are the *head* and *tail* selectors. This program visits the given list twice, a first time for its reversal (using *rev*) and a second time for testing equality (using *eqlist*). We look for an improved program which does not make these two visits. By unfolding we get:

6. $palin([\,]) = true$
7. $palin(a{:}l) = (a = hd(rev(a{:}l)))\; and\; eqlist(l, tl(rev(a{:}l)))$

When trying to fold the r.h.s. of Equation 7 using Equation 1 we have a mismatch between the two expressions $eqlist(l, rev(l))$ and $eqlist(l, tl(rev(a{:}l)))$. We apply higher order generalization to Equation 7 and we have:

8. $palin(a{:}l) \;=\; eqlist(a{:}l, rev(a{:}l)) = \{\text{higher order generalization}\} =$
$\qquad\qquad = (\lambda x.\, eqlist(a{:}l, x))\; rev(a{:}l)$

where the mismatching subexpression has been promoted to the bound variable x. Now, both $\lambda x.\, eqlist(a : l, x)$ and $rev(a : l)$ visit the same data structure $a : l$. We perform a tupling step as suggested by the tupling strategy, and we define:

9. $Q(l) \;=\; \langle \lambda x.\, \mathit{eqlist}(l,x), \mathit{rev}(l)\rangle$

whose recursive equations are as follows:

10. $Q([\,]) = \langle \lambda x.\, \mathit{eqlist}([\,],x), \mathit{rev}([\,])\rangle = \{\text{unfolding}\} = \langle \lambda x.\, \mathit{null}(x), [\,]\rangle$
11. $Q(a{:}l) = \langle \lambda x.\, \mathit{eqlist}(a{:}l,x), \mathit{rev}(a{:}l)\rangle = \{\text{unfolding}\} =$
$= \langle \lambda x.\, (a = \mathit{hd}(x)) \;\mathit{and}\; \mathit{eqlist}(l, \mathit{tl}(x)), \mathit{rev}(l) :: [a]\rangle = \{\text{folding}\} =$
$= \langle \lambda x.\, (a = \mathit{hd}(x)) \;\mathit{and}\; u(\mathit{tl}(x)), v :: [a]\rangle$ where $\langle u,v\rangle = Q(l)$

Now we can fold Equation 1 using Equation 9 and we get:

12. $\mathit{palin}(l) = u(v)$ where $\langle u,v\rangle = Q(l)$

The final program made out of Equations 12, 10, and 11, visits the input list only once in the sense that $Q(a{:}l)$ is defined in terms of $Q(l)$ only. □

However, there may be some drawbacks in applying generalization steps and one should use generalization with parsimony. Indeed, when an expression is generalized to a variable, we loose information about the structure of the generalized expression and that loss may prevent some further improvements. Consider, for instance, Equation 7 of Example 3. The most specific common generalization of the two mismatching expressions in Equation 7 and Equation 1 which did not allow us to perform a folding step, leads to the introduction of the new function d defined as follows: $d(l_1, l_2) = \mathit{eqlist}(l_1, l_2)$. But, unfortunately, in this definition we have now lost the important information that the second argument of d is the reversal of the first argument, while we will use the function d only for arguments satisfying this constraint. Obviously, for computing the function d we cannot hope for a better program than the one provided by Equations 2 and 3, and thus, by using d there is no hope of deriving an efficient program for *palin*. This example also shows the superiority of the higher order generalization over the familiar generalization from expressions to variables. Indeed, as the reader may verify, if one introduces the function

9′. $Q'(l,x) \;=\; \langle \mathit{eqlist}(l,x), \mathit{rev}(l)\rangle$

then the final program one derives, visits the input list twice in the call-by-value mode of evaluation.

Other generalizations may require some form of ‘reflection’ on the symbolic computation constructed so far. For instance, in the supercompilation approach during the construction of the symbolic computation model, the generalization steps can be suggested by an analysis of the part of the computation model already constructed. This analysis makes use of so called *walk grammars* and *meta-transition systems* to reason about *computation histories* in a given model [54]. The analysis may be used, in particular, for avoiding the risk of generalizing ‘too early’ (see, for instance, [53]). Related works are the ones concerned with program improvement based on the analysis of computation histories for which the reader may refer to [1] where a recursion removal technique is described.

Among other forms of reflection on symbolic computations, let us now mention the one related to the *introduction of new operators*. This technique consists in the promotion of a sequence of computations to an independent procedure.

For instance, as shown in the following example, a sequence of additions can be promoted to a single multiplication.

Example 4. [*Introducing New Operators*] Let us consider the Fibonacci program of Section 2 and let us consider the *transformation tree* model [25]. From Equation 3 we get:

$$\begin{aligned} 7.\ fib(k) &= fib(k-1) + fib(k-2) = \{\text{unfolding}\} = \\ &= fib(k-2) + fib(k-3) + fib(k-2) = \{\text{introducing multiplication}\} = \\ &= 2\ fib(k-2) + fib(k-3) = \{\text{unfolding}\} = \\ &= 3\ fib(k-3) + 2\ fib(k-4) = \ldots \end{aligned}$$

Let us now assume that we have discovered the following regularity valid for any $n, k \geq 0$:

$$8.\ fib(k+n+2) = fib(n+1)\ fib(k+1) + fib(n)\ fib(k)$$

This regularity comes from the observation, which may be hard to make in a mechanical way, that when constructing the transformation tree for $fib(k)$, the multiplicative constants in the two summands are values of the *fib* function itself (see also [39]). By the unfold/fold technique we are now able to derive the program made out of the following equations, together with Equations 1 and 2 [39]:

$$\begin{aligned} 9.\ fib(2k+2) &= fib(k+1)^2 + fib(k)^2 \\ 10.\ fib(2k+3) &= fib(k+1)\ fib(k+2) + fib(k)\ fib(k+1) = \{\text{unfolding}\} = \\ &= fib(k+1)\ (fib(k+1) + fib(k)) + fib(k)\ fib(k+1) \end{aligned}$$

and they hold for any $k \geq 0$. Now we may discover one more regularity, namely, the fact that in the m-dag of this last program there is a progressive sequence of cuts, each of them being made out of two consecutive calls of *fib*. Thus, we apply the tupling strategy and we introduce the following function, defined for any $k \geq 0$:

$$11.\ p(k) = \langle fib(k+1), fib(k) \rangle$$

By applying the unfold/fold technique we can then derive the explicit definition of the function $p(k)$ and the following final program:

$$\begin{aligned} &1.\ fib(0) = 1 \\ &2.\ fib(1) = 1 \\ &12.\ fib(k+2) = a + b \quad \text{where } \langle a, b \rangle = p(k) \\ &13.\ p(0) = \langle 1, 1 \rangle \\ &14.\ p(1) = \langle 2, 1 \rangle \\ &15.\ p(2k+2) = \langle a^2 + 2ab, a^2 + b^2 \rangle \quad \text{where } \langle a, b \rangle = p(k) \\ &16.\ p(2k+3) = \langle a^2 + (a+b)^2, a^2 + 2ab \rangle \quad \text{where } \langle a, b \rangle = p(k) \end{aligned}$$

where Equations 12, 15, and 16 hold for any $k \geq 0$. This program is very efficient and takes only $O(\log(n))$ arithmetic operations for computing $fib(n)$ (see also [39]). □

In the partial evaluation field, researchers have studied a method for improving program efficiency which we may classify under the global control issues. This method, called *polyvariant specialization* [9], specializes programs, instead of generalizing them. Indeed, it allows for the generation of various different versions of the same program with the objective of achieving higher performances. The improvement of performances comes from the fact that having more information about the inputs to the program (or function call) one can make some more simplifications at compile time. This method can be viewed as the opposite to generalization, by which one constructs a general program to compute several distinct, but similar functions. Unfortunately, there is no general theory which for any given program tells us when it is better to specialize or to generalize.

There is an inherent limitation in looking for an optimal strategy of when and where to perform specialization and/or generalization steps. Indeed, one cannot hope to construct a universal technique for finding a suitable regularity whenever there is one, which allows us to improve any given program, because the equivalence of two functions can be expressed as a regularity of their symbolic computation models, and yet equivalence of functions is undecidable. However, in practice, regularities which are useful for program transformation, are often decidable properties, and they can also be found by means of efficient algorithms.

7 Relating the Three Step Program Transformation Methodology to Program Specialization, Deforestation, and Finite Differencing

The three steps of the general program transformation methodology we have presented in the previous sections do not always refer to a definite sequence of actions performed when applying a particular program transformation technique. For supercompilation, unfold/fold transformation, generalized partial computation, compiling control, and partial deduction, one may easily identify those three steps of the methododlogy. However, for some other techniques, like partial evaluation of functional and imperative programs, program specialization, mixed-computation [15], or deforestation [55], it is not always easy to do the same. Nevertheless, we think that the concepts of symbolic computation, search for regularities, and extraction of final programs, are to some extent present in those techniques as well.

To see this, we would like to report the following phrases taken from [26, pages 68–69]:

"Our main thesis is that program specialization can be done in three steps.

1. Given the value of part of the program's input, obtain a description of all computational states reachable when running the program on all possible input values.
2. Redefine the program's control by incorporating parts of the data state into the control state, yielding perhaps several specialized versions of each of the program's control points (0, 1, or more; hence the term *polyvariant* specialization).

3. The resulting program usually contains many trivial transitions. Optimize it by traditional techniques, yielding the specialized (or *residual*) program."

This description of the three steps which underline most program specialization techniques, including partial evaluation and mixed computation, matches quite closely the three steps of the methodology we have presented in this paper.

The first step of the program specialization methodology corresponds to our first two steps, that is, the generation of a symbolic computation process and the search for regularities of this symbolic computation. More precisely, as we have shown in Section 3, symbolic computation can be used to 'obtain the description of all computational states reachable when running the program on all possible input values'. By finding suitable regularities we may make sure that this description is *finite*.

In practice, however, many specialization techniques use symbolic computation models based on abstract interpretation which, unlike the models considered in this paper, cannot always be described in terms of an unfolding process. Among these abstract interpretation-based techniques we would like to mention the techniques for *binding time analysis* [27, 34] and the *regular approximation* techniques for approximating the least Herbrand model of a logic program [23].

The second and third steps of the program specialization methodology correspond to what we have called here 'extraction of the new program'. In this paper we have only pointed out the derivation techniques which substantially change the program's control and we have not given much attention to various post-processing techniques (see, for instance, the renaming techniques in the compiling control example and in the partial deduction example presented in the previous sections). These post-processing techniques can be considered to be part of Step 3 of program specialization.

The reader may notice that program specialization is a particular instance of the general program methodology we presented in this paper, because it is idempotent [54], in the sense that when specializing a program which has been already specialized, we get the same program we derived after the first specialization. Other transformation methods, such as supercompilation and rule-based program transformation are not idempotent. This fact can be illustrated by Example 4 where after deriving the program made out of Equations 1, 2, 9, and 10, by discovering a new regularity we were able to derive a new and more efficient program.

Some elements of the general methodology based on symbolic computation, search for regularities, and extraction of final programs are also present in the deforestation technique.

Deforestation is designed to eliminate intermediate data structures from functional programs by introducing new function definitions which are equivalent to the composition of already available functions. In this sense deforestation can be viewed as an instance of the *composition strategy* (also called *fusion*) introduced in the field of unfold/fold transformation [10, 18] and it is also closely related to Scherlis' *internal specialization* [46] and supercompilation [48].

Deforestation works via generating, by an unfolding process, from an initial term containing nested function calls other (possibly infinitely many) terms. Although deforestation does not explicitly construct any symbolic computation model, the unfolding steps it requires can be viewed as a symbolic computation. The idea of finding regularities by identifying similar configurations is also present. In particular, the deforestation algorithm terminates only if a finite number of terms modulo variants, is generated during unfolding. If this is the case then it is possible to avoid intermediate data structures by introducing a finite number of new function definitions which correspond to (a subset of) the terms generated by unfolding.

Finally, among other techniques for program derivation we want to consider also *finite differencing* [35, 36]. The revisitation of this technique as an instance of our general methodology is not very straightforward. However, the three steps of finite differencing, which are:

> "i) syntactic recognition of computational bottlenecks appearing within a program P, ii) choosing invariants whose maintenance inside P allows these bottlenecks to be removed, and iii) scheduling how collections of invariants can be maintained in P" [35, page 40]

correspond, respectively, to: i) the symbolic computation model which allows for the detection of the bottlenecks, ii) the search for regularities which are the invariants to be maintained, and iii) the program extraction by which new sequences of operations are generated with the objective of maintaining those invariants. The reader may find more information about finite differencing in the cited papers.

8 Correspondences Among Some Program Transformation Techniques

We will not present in details the formal relationships and correspondences among the many program transformation techniques mentioned in this paper, because as we already said, these correspondences can be considered as 'common knowledge' of the people working in the field. Let us simply mention among some other similar results, the following ones: i) the unfold/fold view of the mixed computation technique described in [16], ii) the equivalence of driving in supercompilation and partial deduction shown in [25] for a particular class of programs, and iii) the straightforward way of using the unfold/fold transformation technique to simulate partial deduction [44]. We now present this simulation in a simple example.

Example 5. Suppose we want to partially evaluate the following program:

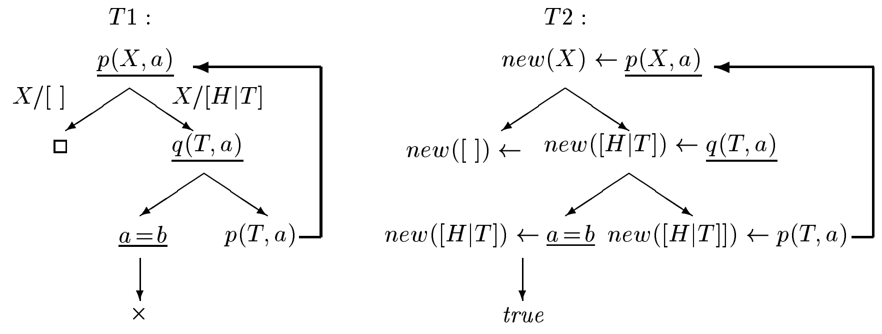
$p([\,], Y) \leftarrow$
$p([H|T], Y) \leftarrow q(T, Y)$
$q(T, Y) \leftarrow Y = b$

$q(T, Y) \leftarrow p(T, Y)$

with respect to the set $\{p(X, a)\}$. We follow the partial deduction technique as proposed in [31] and we get the initial portion of the SLDNF-tree $T1$ depicted in Figure 6. By considering the non-failing branches of that tree and taking the corresponding resultants, we get the program $P1$:

$p([\,], a) \leftarrow$
$p([H|T], a) \leftarrow p(T, a)$

which is a correct partial deduction because the requirements for independence and closedness are satisfied. Indeed, i) independence is a trivial consequence of the fact that in the set $\{p(X, a)\}$ there is one atom only, and ii) closedness is a consequence of the fact that the atoms $p([\,], a)$, $p([H|T], a)$, and $p(T, a)$ are all instances of $p(X, a)$.



$T1$:
$\underline{p(X, a)}$
$X/[\,]$
$X/[H|T]$
$\square$
$\underline{q(T, a)}$
$\underline{a = b}$
$p(T, a)$
$\times$
$T2$:
$new(X) \leftarrow \underline{p(X, a)}$
$new([\,]) \leftarrow$
$new([H|T]) \leftarrow \underline{q(T, a)}$
$new([H|T]) \leftarrow \underline{a = b}$
$new([H|T]]) \leftarrow p(T, a)$
$true$

Fig. 6. An SLDNF-tree for partial deduction ($T1$) and the corresponding unfolding tree ($T2$). Underlined goals are the unfolded ones. Upgoing arrows relate similar nodes.

Using the unfold/fold method we first introduce a clause whose body is made out of the goal $p(X, a)$ and whose head has a fresh predicate symbol, say *newp*. The arguments of the head are the variables occurring in the body. Thus, we introduce the clause:

$newp(X) \leftarrow p(X, a)$

Then, by using the unfold/fold method, we derive a program which can be used for evaluating queries of the form: $\leftarrow newp(X)$, instead of $\leftarrow p(X, a)$. The derivation process takes the form of the unfolding tree $T2$ depicted in Figure 6. From that tree we can extract the following program $P2$ by performing, as we have indicated in Section 5, a final folding step (whereby the body $p(T, a)$ is replaced by $newp(T)$):

$newp([\,]) \leftarrow$
$newp([H|T]) \leftarrow newp(T)$

which has performances similar (actually, higher, because *newp* has one argument only) to those of $P1$. $\square$

The reader should notice the perfect correspondence between partial deduction and the unfold/fold technique we have now illustrated. In particular, we want to stress that the condition which allowed us to perform the final folding step during program extraction, that is, the fact that $p(T, a)$ is an instance of the body of the clause $newp(X) \leftarrow p(X, a)$, is exactly the same condition, that is, closedness, which ensures the correctness of the partial deduction process [31].

9 Conclusions

We have presented a general methodology for the derivation of programs which underlines some familiar program transformation techniques like, for instance, partial evaluation, supercompilation, rule-based program derivation, program specialization, and compiling control. This methodology can often be mechanized, although the extent to which this mechanization is possible, very much depends on the technique under consideration.

This methodology is made out of three steps. They are: i) the construction of the symbolic computation model, ii) the search for regularities in that model, and iii) the extraction of the new program. Through the presentation of these steps and some examples, we have illustrated in an informal way the correspondences among the above-mentioned program transformation techniques. They are all based on the construction of some sort of finite directed graphs whose arcs represent either the steps of the computations or the similarity relations among computation states (or configurations). We have also briefly considered the so called control issue. It is related to the problem of guiding the actions to be performed during the three steps of the methodology, and in particular we have looked at various forms of the generalization strategy.

In this paper we have stressed the similarities among the various techniques for program transformation. There are, however, also many differences among them. They are due, besides other reasons, to the different languages and the different semantics which are considered, and to the degree of automation which is required for their implementation. For instance, in the case of partial evaluation one strives for a completely automated process, whereas in supercompilation and rule-based program derivation, one also allows for interactive theorem proving capabilities.

Acknowledgements

We have profited from many conversations with several people over the past years. In particular, we would like to mention R. S. Bird, M. Bruynooghe, R. M. Burstall, W.-N. Chin, J. Darlington, D. De Schreye, A. P. Ershov, M. S. Feather, Y. Futamura, J. Gallagher, R. Glück, N. D. Jones, T. Mogensen, B. Möller, H. Partsch, R. Paige, P. Pepper, W. L. Scherlis, V. Turchin, P. Wadler, and our colleagues of the IFIP W.G. 2.1, the Italian Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, and Compulog II Esprit Project. We thank S. Renault and M. H. Sørensen for comments on a draft of this paper.

References

1. J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.
2. R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.
3. R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–418, 1980.
4. R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Toplas*, 6(4):487–504, 1984.
5. D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988. IFIP TC2 Workshop on Partial and Mixed Computation, Gammel Avernæs, Denmark, 1987.
6. M. Bruynooghe and D. Boulanger. Abstract interpretation for (constraint) logic programming. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, NATO ASI Series F, Vol. 131, pages 228–260. Springer-Verlag, 1994.
7. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.
8. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. *New Generation Computing*, 11:47–79, 1992.
9. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.
10. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
11. W.-N. Chin. Safe fusion of functional expressions. In *Proceedings of ACM SIGPLAN Symposium on Lisp and Functional Programming, San Francisco, Calif., U.S.A.*, pages 11–20. ACM Press, 1992.
12. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings 20th ACM SIGPLAN-SIGACT Symposium on Princples of Programming Languages (POPL '93)*, pages 493–501. ACM Press, 1993.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings 4th ACM-SIGPLAN Symposium on Princples of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.
14. J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.
15. A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.
16. A. P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18(1):41–67, 1982.
17. A.P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *Special Issue of New Generation Computing: Workshop on Partial Evaluation and Mixed Computation*, volume 6, Nos. 2&3. Ohmsha Ltd. and Springer-Verlag, 1988.
18. M. S. Feather. A survey and classification of some program transformation techniques. In L. G. L. T. Meertens, editor, *Proceedings IFIP TC2 Working Conference*

on Program Specification and Transformation, Bad Tölz, Germany, pages 165–195. North-Holland, 1987.

19. Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90:61–79, 1991.
20. J. P. Gallagher. Transforming programs by specializing interpreters. In *Proceedings Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 109–122, 1986.
21. J. P. Gallagher. Tutorial on specialization of logic programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.
22. J. P. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 6(2):305–333, 1991.
23. J. P. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs and their uses. In *Proceedings of the 11th International Conference on Logic Programming, ICLP'94*, pages 599–613. MIT Press, 1994.
24. R. Glück. On the mechanics of metasystem hierarchies in program transformation. In M. Proietti, editor, *Proceedings of LOPSTR '95, Utrecht, The Netherlands*, Lecture Notes in Computer Science 1048, pages 234–251. Springer-Verlag, 1996.
25. R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *International Symposium on Programming Language Implementation and Logic Programming, PLILP '94*, Lecture Notes in Computer Science 844, pages 165–181. Springer-Verlag, 1994.
26. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
27. 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.
28. L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.
29. R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22 (7):424–436, 1979.
30. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In *Proceedings of LOPSTR '95, Utrecht, The Netherlands*, Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1996.
31. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
32. K. Marriott, H. Søndergaard, and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
33. L. Naish. *Negation and Control in Prolog.* Lecture Notes in Computer Science 238. Springer-Verlag, 1985.
34. H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.
35. R. Paige. Symbolic finite differencing - Part I. In N. D. Jones, editor, *Third European Symposium on Programming, ESOP '90*, Lecture Notes in Computer Science 432, pages 36–56. Springer-Verlag, 1990.
36. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

37. R. Paige, J. Reif, and R. Wachter, editors. *Parallel Algorithm Derivation and Program Transformation, Proc. Workshop at Courant Institute of Mathematical Sciences, New York, USA, 1991.* Kluwer Academic Publishers, 1993.
38. A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Symposium on Lisp and Functional Programming*, pages 273–281. ACM Press, 1984.
39. A. Pettorossi and R. M. Burstall. Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica*, 18:181–206, 1982.
40. A. Pettorossi and M. Proietti. Rules and strategies for program transformation. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development, IFIP TC2/W.G. 2.1 State-of-the-Art Report*, Lecture Notes in Computer Science 755, pages 263–304. Springer-Verlag, 1993.
41. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
42. A. Pettorossi and A. Skowron. The lambda abstraction strategy for program derivation. *Fundamenta Informaticae*, XII(4):541–561, 1989.
43. M. Proietti and A. Pettorossi. Synthesis of eureka predicates for developing logic programs. In N. D. Jones, editor, *Third European Symposium on Programming, ESOP '90*, Lecture Notes in Computer Science 432, pages 306–325. Springer-Verlag, 1990.
44. M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16(1–2):123–161, 1993.
45. M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, *Logic Program Synthesis and Transformation, Proceedings of LOPSTR '93, Louvain-la-Neuve, Belgium*, Workshops in Computing, pages 141–158. Springer-Verlag, 1994.
46. W. L. Scherlis. Program improvement by internal specialization. In *Proc. 8th ACM Symposium on Principles of Programming Languages, Williamsburgh, Va*, pages 41–49. ACM Press, 1981.
47. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 465–479. MIT Press, 1995.
48. M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Fifth European Symposium on Programming Languages and Systems, ESOP '94*, Lecture Notes in Computer Science 788, pages 485–500. Springer-Verlag, 1994.
49. L. S. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1994. Second Edition.
50. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings Second International Conference on Logic Programming, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.
51. V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *Proceedings of 7th Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 85, pages 645–657. Springer-Verlag, 1980.
52. V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

53. V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation, Proc. of the IFIP TC2 Working Conference, Gammel Avernæs, Denmark, 1987*. North-Holland, 1988.
54. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
55. P. L. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of ESOP '88*, Lecture Notes in Computer Science 300, pages 344–358. Springer-Verlag, 1988.
56. H. Zhu. How powerful are folding / unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, 1994.