Regular Approximation of Computation Paths in Logic and Functional Languages

John Gallagher   Laura Lafave

March 1996   **CSTR-96-001**

University of Bristol

Department of Computer Science

Also issued as ACRC-96:CS-001

Regular Approximation of Computation Paths in Logic and Functional Languages

John Gallagher and Laura Lafave

Department of Computer Science, University of Bristol, Bristol BS8 1TR, U.K.
{john,lafave}@cs.bris.ac.uk

Abstract. The aim of this work is to compute descriptions of successful computation paths in logic or functional program executions. Computation paths are represented as terms, built from special constructor symbols, each constructor symbol corresponding to a specific clause or equation in a program. Such terms, called *trace-terms*, are abstractions of computation trees, which capture information about the control flow of the program. A method of approximating trace-terms is described, based on well-established methods for computing regular approximations of terms. The special function symbols are first introduced into programs as extra arguments in predicates or functions. Then a regular approximation of the program is computed, describing the terms occurring in some set of program executions. The approximation of the extra arguments (the trace-terms) can then be examined to see what computation paths were followed during the computation. This information can then be used to control both off-line or on-line specialisation systems. A key aspect of the analysis is the use of suitable *widening* operations during the regular approximation, in order to preserve information on determinacy and branching structure of the computation. This method is applicable to both logic and functional languages, and appears to offer appropriate control information in both formalisms.

1 Introduction

Information about the control flow of program is useful for guiding partial evaluation. In off-line partial evaluation methods, control information is collected during a separate analysis phase; this is then used to drive the specialisation phase. In on-line approaches, control information is gathered dynamically while partial evaluation is actually being performed. The aim of this paper is to compute descriptions of successful computation paths in logic or functional program executions. An approach covering both logic and functional programs is sought, so as to uncover the common principles governing specialisation in both formalisms.

The proposal put forward in this paper is to try to capture the *shape* of computations, including the branching, looping, determinacy and non-determinacy in a set of computations. This information is useful in making decisions about control and polyvariance.

Computation paths are represented as terms, built from special constructor symbols, each constructor symbol corresponding to a specific clause or equation in a program. Such terms, called *trace-terms*, are abstractions of computation trees, which capture information about the control flow of the program. A method of approximating trace-terms will be described, based on well-established methods for computing regular approximations of terms.

Trace-terms could also be used in conjunction with existing methods of generating control information, especially termination analyses. The paper contains examples of combining trace-terms with *process trees* [SG95b] or *m-trees* [MG95].

Related ideas are already in the literature; these include *neighbourhoods* based on computational *histories* [Tur88] and *characteristic trees* [GB91, Leu95]. The aim of both these approaches is to use computational behaviour as an abstraction: two atoms or terms that give rise to similar computations are regarded as identical from the point of view of partial evaluation.

Trace-terms seem to offer a refinement of these notions, allowing more flexible treatment and more precise control than previously. The idea of regular approximation of trace-terms also appears to be related to more recent work by Turchin on *walk grammars* [Tur93, Tur96], though the exact connections have still to be studied. Trace-terms may offer a uniform treatment of all these ideas.

2 Representing Computation Paths

In this section we introduce a representation of (successful) computation paths. The idea is very similar in logic and functional programming, but some of the formal details differ. The idea is to record the clauses or statements used at each step in a computation. We wish to do this independently of any particular computation rule or reduction strategy. In logic programming we may think of this approximately as an AND-parallel computation where all atoms in a goal are simultaneously resolved upon. We can define an AND-tree capturing this directly. In functional programming this is not directly possible, but we can extract order-independent representations from a computation that was constructed using a specific reduction strategy.

2.1 Trace-terms in Logic Program Computations

The idea of using traces of logic program derivations has appeared in various forms, e.g. [Gal86], [Sha87], [GB91] but these representations employed sequences of clauses used in SLD derivations. Thus the SLD computation rule was also encoded in these representations. The method to be described here is independent of the computation rule, and this has certain advantages as will be seen.

Definition 1. clause identifiers

Let P be a definite program. Let $\{c_1, \ldots, c_n\}$ be the set of clauses in P. Let a_j, $0 \leq j \leq n$ be the number of atoms in the body of the j^{th} clause. Let $\{\varphi_1/a_1, \ldots, \varphi_n/a_n\}$ be a set of n distinct functors (not in the language of P), where for each j, φ_j is a functor of arity a_j. That is, each clause in P is associated with a functor whose arity is equal to the number of atoms in the body of the clause. These functors will be called *clause identifiers.*

These functors will be used to represent the structure of successful computations. A successful computation in a definite logic program P is represented as an AND-tree, defined as follows.

Definition 2. AND-tree
An AND-tree (for program P) is a tree each of whose nodes is labelled by an atom and a clause, such that

1. each non-leaf node is labelled by a clause $A \leftarrow A_1, \ldots, A_k$ and an atom $A\theta$ (for some substitution θ), and has children $A_1\theta, \ldots, A_k\theta$,
2. each leaf node is labelled by a clause $A \leftarrow \mathit{true}$ and an atom $A\theta$ (for some θ).

Lemma 3. *(Stärk)*
Let P be a program and $\leftarrow A$ be a goal. Then θ is a correct answer for $P \cup \{\leftarrow A\}$ if and only if there is an AND-tree (for P) with root node labelled by $A\theta$.

Furthermore, a successful SLD-derivation with computation rule R can be transformed into an AND-tree. Since AND-trees are independent of the computation rule, AND-trees offer a more abstract characterisation of successful computations than SLD derivations.

Each AND-tree can be associated with a term constructed entirely from clause identifiers.

Definition 4. trace-term corresponding to an AND-tree
Let T be an AND-tree; define $\alpha(T)$ to be either

1. φ_j, if T is a single leaf node labelled by the unit clause identified by φ_j; or
2. $\varphi_i(\alpha(T_1), \ldots, \alpha(T_{a_i}))$, if T is labelled by the clause identified by φ_i/a_i, and has immediate subtrees $T_1, \ldots, T_{a_i}$.

Note that α may map more than one AND-tree to the same term. In fact, the main idea is to use the term $\alpha(T)$ as an abstraction of the AND-tree T. The idea is thus similar in its aims to the idea of *histories* [Tur88], and, more closely, *characteristic paths and trees* [GB91], [Leu95]. The main advantage of this formulation over the techniques just mentioned is that it abstracts away the computation rule, and it is in a much more convenient form for performing analysis of computations, since we can adapt well-established techniques for approximating term structures to approximate terms representing AND-trees. Trace-terms differ from characteristic trees also in that they represent complete answer traces rather than partial unfolding traces.
Example: Let P be the following program:

```
rev([],[]) <- true.
rev([X|Xs],Ys) <- rev(Xs,Zs), append(Zs,[X],Ys).

append([],Ys,Ys) <- true.
append([X|Xs],Ys,[X|Zs]) <- append(Xs,Ys,Zs).
```

Assign the functions $rev1/0, rev2/2, app1/0, app2/1$ to the above four clauses respectively. Consider the goal `<- rev([a,b],W)`. The trace-term for the computation of this goal is $rev2(rev2(rev1, app1), app2(app1))$. Note that the computation of the goal `<- rev([c,d],W)` would have exactly the same trace-term, since the list elements play no role in the control.

Now consider the goal `<- append(U,V,[a,b])`. This is non-deterministic and has a set of trace-terms, namely $\{app1, app2(app1), app2(app2(app1))\}$ each of which represents a successful computation.

3 Incorporating Trace-terms in Logic Programs

Trace-terms can easily be added to logic programs, so that the computation returns a trace term as well as its normal result. Let P be a program and let the i^{th} clause be $p(\bar{t}) \leftarrow q_1(\bar{t_1}), \ldots, q_{a_i}(\bar{t}_{a_i})$. Let φ_i/a_i be the trace function symbol associated with the i^{th} clause. Transform each such clause to $p(\bar{t}, \varphi(y_1, \ldots, y_{a_i})) \leftarrow q_1(\bar{t_1}, y_1), \ldots, q_{a_i}(\bar{t}_{a_i}, y_{a_i})$, where $y_1, \ldots, y_{a_i}$ are distinct variables not occurring elsewhere in the clause.

Finally, transform each atomic goal $\leftarrow q(\bar{s})$ to $\leftarrow q(\bar{s}, w)$, where w is a variable not occurring elsewhere in the goal. (Consider only atomic goals without loss of generality).

Example: let P be the reverse program, as in the previous example. It is transformed into the following clauses.

```
rev([],[],rev1) <- true.
rev([X|Xs],Ys,rev2(Y1,Y2)) <- rev(Xs,Zs,Y1), append(Zs,[X],Ys,Y2).

append([],Ys,Ys,app1) <- true.
append([X|Xs],Ys,[X|Zs],app2(Y1)) <- append(Xs,Ys,Zs, Y1).
```

The goal `<- reverse([a,b],W)` becomes `<- reverse([a,b],W,U)`, which returns the answer `W = [b,a]`, `U = rev2(rev2(rev1, app1),app2(app1))`. Similarly the goal `append(U,V,[a,b],Y)` returns three answers, where `Y` returns the appropriate trace-term corresponding to each answer.

It is obvious that the transformation has no effect on the answers computed for the original program.

Note also that the trace-term can be used to drive the computation as well as record it, since the trace-term uniquely identifies the clause to be resolved at each step. The goal `reverse(X,Y,rev2(rev2(rev1, app1),app2(app1)))` returns the answer `X = [U,V], Y = [V,U]`, which is the most general substitution associated with that trace-term.

3.1 Trace Terms in Functional Programming

The use of trace-terms to represent computations, and give control information in both logic and functional programming, underlines the connections between specialisation in logic and functional programming. Such connections have already been pointed out by Jones, Glück and Sørensen in their work on supercompilation, driving, deforestation and partial evaluation [SGJ94, SG95b, GS94]. Positive driving, a variant of driving developed by Glück and Sørensen, has been shown to be equivalent to partial deduction in logic languages.

The Simple Functional Language $\mathcal{M}$ In order to discuss representing functional computations precisely, we will employ a simple functional language $\mathcal{M}$ [GK93, SGJ94, GS94, SG95b].

Definition 5. Language $\mathcal{M}$

$$\begin{array}{rl}
d ::= & f\, v_1 \dots v_n \to t \\
\mid & g\, p_1\, v_1 \dots v_n \to t_1 \\
& \vdots \\
& g\, p_m\, v_1 \dots v_n \to t_m \\
t ::= & b \mid f\, b_1 \dots b_n \mid g\, t\, b_1 \dots b_n \mid \mathbf{if}\ b_1 = b_2\ \mathbf{then}\, t_1\ \mathbf{else}\, t_2 \\
b ::= & v \mid c\, b_1 \dots b_n \\
p ::= & c\, v_1 \dots v_n
\end{array}$$

The following restrictions exist for this language:

- Function definitions may have no more than one argument defined on patterns, which must be non-nested and linear.
- Function calls cannot occur as arguments in other function calls unless the function call is the first argument of a g-function call.
- All variables in the right side of a definition occur on the left side and the left side of a definition is linear (i.e. no variable occurs more than once).

In addition, it is clear from the definition of the language that functions are not permitted to be arguments of a constructor. These restrictions of the language guarantee that the results of computations are independent of the order of evaluation. Finally, for the language $\mathcal{M}$ we will specify the data structure for lists, an infix notation $a : as$ to denote the concatenation of a with as, and the terms $[], [x_1, \dots, x_n]$ to represent Nil and $x_1 : \dots : x_n : Nil$ respectively.

Incorporating Trace Terms with Computations In order to incorporate trace terms into these programs, we create constructor terms to represent the statements in the program.

Definition 6. statement identifiers

Let p be a program in the language $\mathcal{M}$. Let $\{s_1, \dots, s_n\}$ be the set of statements in p defining a function h. Associate distinct constructors of the form

φ/a (where a is the arity of φ), not in the language of $\mathcal{M}$, to the statements, according to the following two rules:

- If $s_i = h\,x_1, \ldots, x_n \rightarrow$ **if** $b = b'$ **then** t **else** t', then associate with s_i the constructors φ_i^T/a_i^T and φ_i^F/a_i^F, such that a_i^T is the number of function occurrences in t, and a_i^F is the number of function occurrences in t'. φ_i^T/a_i^T and φ_i^F/a_i^F are associated with the **then** and **else** branches respectively of the conditional statement.
- Otherwise, if the right hand side of s_i is either a constructor term, a f-function or a g-function, associate with s_i the constructor φ_i/a_i where a_i is the number of function occurrences in the right hand side of s_i.

These constructors will be called *statement identifiers.*

For example, examine the program which finds the last element of a list:

$$\begin{array}{lll} last\,(x : xs) & \rightarrow check\; xs\; x & < last1/1 > \\ check\,[]\,x & \rightarrow x & < check1/0 > \\ check\,(z : zs)\,x & \rightarrow check\; zs\; z & < check2/1 > \end{array}$$

For the ground term $last(A : B)$, the computation is:

$$last\,(A : B) \rightarrow check\; B\; A \rightarrow check\; []\; B \rightarrow B$$

We assign three statement identifiers $last1/1$, $check1/0$, and $check2/1$ to each of the statements in the program, as noted to the right of the program.

Definition 7. labelled computation
A *labelled computation* S' for the computation $S = t_0, \ldots, t_m$, $m > 0$, is a sequence of ordered pairs $(t_j, \varphi_k/a_k)$ where t_j is reduced to t_{j+1} in S using the statement (or conditional branch of the statement) identified by φ_k/a_k and the last member of the sequence is (t_m, Nil) where Nil indicates the empty statement identifier.

We now define a trace-term associated with a labelled computation. The trace-term should be independent of the computation rule, just as in the logic programming case. There is nothing in functional programs directly equivalent to the AND-tree of logic programs, so the contruction of the trace-term is not quite so direct.

Let $S = (t_0, si_0), \ldots, (t_m, Nil)$, $m > 0$, be a computation where t_m contains no function symbols. During the computation, every function occurring in a term in S is reduced at some point in the computation. Let f be (an occurrence of) a function in t_j; that occurrence appears in $t_j, t_{j+1}, t_{j+2}, \ldots$ until some term t_{j+d} $(d \geq 0)$, at which that occurrence of f is reduced. We define $red(f)$ to be $j + d$ in this case. That is, $red(f)$ is the index of the step at which f is reduced in the computation. For every such occurrence of a function symbol in S, red is uniquely defined. Notice that the restrictions on $\mathcal{M}$ ensure that red is well-defined.

Definition 8. Let $S = (t_0, si_0), \ldots, (t_m, \mathit{Nil})$ be a labelled computation where t_m contains no function symbols.

Define a trace term associated with each functional subterm occurring in S, by induction on the length of the computation.

1. Let $m = 1$. The only functional term is $t_0 = t_{m-1}$. The trace term associated with t_{m-1} is $\varphi_m/0$ (this must have arity 0 since otherwise t_m would contains a function symbol).
2. If $m > 1$, let $f\, t_1 \ldots t_s$ be a subterm occurring in t_k, $k < m - 1$. Let $red(f) = k + d$, $d \geq 0$. Suppose the $k + d^{th}$ pair in S is $(t_{k+d}, \varphi_{k+d}/a_{k+d})$, where $t_{k+d} = e[r]$ and $r = f\, t'_1 \ldots t'_s$. Suppose the right hand side, **then**, or **else** part (as appropriate) of the statement identified by φ_{k+d}/a_{k+d} is $\bar{t}$. Then t_{k+d+1} is $e[\bar{t}\theta]$ for some substitution θ. Suppose $\langle \beta_1, \ldots, \beta_{a_{k+d}} \rangle$ is the tuple of trace-terms associated with the functional terms in $\bar{t}\theta$. Then the trace-term associated with the original term $ft_1 \ldots t_s$ occurring in t_k is $\varphi_{k+d}(\beta_1, \ldots, \beta_{a_{k+d}})$.

Different labelled computations may have the same trace-term, and thus a trace-term is an abstraction of a computation. Returning to the example program, the labelled computation with root node $t_0 = last(A : B)$ is:

$$(last\,(A : B), last1/1) \rightarrow (check\, B\, A, check2/1) \rightarrow (check\, []\, B, check1/0) \rightarrow (B, \mathit{Nil})$$

and the computed trace term for t_0 is $last1(check2(check1))$, since $t_3 = B$, the trace term associated with $t_2 = check\, []\, B$ is $check1$. Similarly, the trace term associated with $t_1 = check\ B\ A$ is $check2(check1)$, which follows from the definition with $k = 1$ and $d = 0$. Repeating this procedure until t_0 results in the trace term for t_0. Finally, we notes that the independence of trace-terms from the reduction order is not illustrated in this example since all trace-term contructors are unary.

4 Approximation of Sets of Trace-Terms

Once trace-terms are incorporated into the program, they can be treated like any other piece of data. Given a program P and a goal or term t where P and t are transformed to include trace-terms, it is of interest for specialisation purposes to find out the set of all the trace-terms arising from the computation of t in P, since the specialised program need follow only the successful computation paths and can precompute sections of computation that are determined.

In general, there can be an infinite number of distinct trace-terms associated with a goal or term. In fact an infinite number of answers is normal for specialisation since usually the input data needed to drive the computation to termination in a recursive program is missing.

There are a number of static analysis techniques available for computing finite descriptions of infinite sets of terms. Usually, of course, such finite descriptions represent approximations of the infinite set. One such technique which is well-understood and has been successfully implemented is the method of *regular approximation* [GdW94], [HCCar], [BJ92], [Hei92], [FSVY91]. In functional languages, a method for deriving regular tree-automata approximations of programs was developed by Jones [Jon87]. The aim of this analysis technique is to compute for each argument position in the program a regular description of the set of values that can appear at that position in successful computations. Properties of regular descriptions ensure that it is decidable whether a given values is or is not a member of the set that is described.

We describe two approaches to computing an approximation of the set of trace terms associated with a computation.

- Enumerate the set of completed computations using a fixpoint algorithm, with widening to ensure termination.
- Obtain a finite description of the set of possible computations using any other technique, and then extract from it a recursive description of the set of associated trace-terms.

5 Fixpoint Algorithm for Regular Approximation

The first approach is described for the logic programming case. The method used is the procedure for regular approximation of definite programs developed by Gallagher and de Waal [GdW94], which has been used in several experiments [dWG94], [SG95a].

Regular approximations of sets of terms are represented as regular unary logic programs (RUL programs) [YS90], which consist of clauses of the form

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$$

where $x_1, \ldots, x_n$ are distinct variables. A regular approximation of a predicate p/n is an RUL program containing a clause $approx(p(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$. The predicates $t_1(x_1), \ldots, t_n(x_n)$ give the approximations for each of the arguments of p/n.

The algorithm computes a monotonically increasing sequence of RUL programs bottom-up, that is, propagating approximations from the body of a clause to the head. The procedure terminates when a fixed point is found. Termination is ensured by a so-called *widening* procedure, which introduced recursive unary predicates when some "looping" criterion is detected. (Methods of widening seem to be the main difference between the various techniques for regular approximation of logic programs).

5.1 Widening Techniques

As mentioned above, a critical part of the approximation procedure is a suitable *widening* operation. The essence of this operation is that some RUL program is transformed to another RUL program incorporating a new recursive predicate. The new RUL program is "bigger" in the sense that any goal that succeeds with the original RUL program also succeeds in the new one.

The widening rule employed in our current approximation procedure is as follows.

Definition 9. simple widening

Let R be an RUL program containing predicates t_1 and t_2, where t_2 depends on t_1 (that is, t_1 calls t_2, possibly via other predicates). If the set of function symbols in the heads of the clauses defining t_1 and t_2 are the same, then a new recursive predicate, say t_3 is formed, which is the union of t_1 and t_2, and t_3 replaces occurrences of both t_1 and t_2.

Example: Let R be the following RUL program.

```
p(f(X)) <- q(X).              r(f(X)) <- s(X).
p(a) <- true.                 r(a) <- true.

q(g(X)) <- r(X).              s(b) <- true.
```

In R, p depends on r, and both predicates contain the same set of functions ($f/1$ and $a/0$) in their clause heads. Widening is performed, introducing a recursive predicate t.

```
t(f(X)) <- u(X).             u(g(X)) <- t(X).
t(a) <- true.                u(b) <- true.
```

This widening operation ensures an upper bound on the size of RUL programs (over a finite set of function symbols). However, it introduces imprecision into the approximation, which though acceptable for many applications like regular type-checking, reduces its usefulness for program specialisation.

Example: Given the trace-term $app2(app2(app1))$ from the *reverse* program shown above, the corresponding RUL program is

```
t_0(app2(X)) <- t_1(X).
t_1(app2(X)) <- t_2(X).
t_2(app1) <- true.
```

In this program t_0 depends on t_1 and both contain just the function $app2/1$ in their clause heads. Hence widening is applied, giving the recursive RUL program

```
t_3(app2(X)) <- t_3(X).
t_3(app1) <- true.
```

The first RUL program represented a single deterministic computation, while the second represents an infinite set of computations. In other words, a two-element list has been generalised into a list of arbitrary length.

5.2 Improved Widening Operators For Approximating Trace-Terms

Given that one of the main aims of analysing control flow for specialisation is to detect determinacy, another widening operator is now considered. The general aim is to ensure termination of the regular approximation procedure, but to retain as much information as possible about deterministic sections of the computation.

Definition 10. A predicate is called *determinate* if it contains one clause in its definition. It is called *non-determinate* if it contains more than one clause in its definition.

The essential idea of widening operators is to identify a unary predicate which depends on a "similar" predicate. In the simple cases seen so far, the notion of similarity is based on examining the function symbols occurring in the head of the clauses defining the two predicates. The similarity is then strengthened into identity by creating a single recursive predicate to replace both. Refinements of this idea are possible, based on more detailed definitions of "similarity". Similarity based on comparing the function symbols in clause heads can be generalised by looking at the function symbols at a number of levels.

5.3 Similarity with Determinacy to a Fixed Depth

To capture determinacy, we would like to compare the trace-terms defined by regular predicates up to the depth corresponding to the level of determinacy in the predicates. Predicates defined by only one clause are called *determinate predicates*. Let R^0 be an RUL program containing some determinate predicates. Let R^1 be the program obtained by unfolding (once) all calls to determinate predicates in the clause bodies, except self-unfolding (unfolding some body atom using the clause being unfolded). Let R^k be the result (if it is possible) of applying this operation k times. The process may terminate before k is reached, that is, R^m might not contain any unfoldable determinate predicates in clause bodies, for some m (including 0). In this case $R^m = R^{m+1} = \ldots = R^k$.

Example: Let R be the following RUL program.

```
t(f(X)) :- r(X).               s(h(X)) :- q(X).
t(g(X,Y)) :- s(X),s(Y).        q(f(X)) :- p(X).
r(f(X)) :- t(X).               p(a).
```

R^1 and R^2 are shown in Figure 1. There are no determinate predicates in clause bodies of R^2, so all the determinacy has been made explicit.

R^k makes explicit the determinacy in R up to k levels. In R^k, clause heads contain function symbols nested. Given a predicate t, we can speak of the *terms* in the heads of the clauses defining t, rather than just the function symbols as before. A widening operator (parameterised by k) based on this notion is now proposed.

t(f(f(X))) :- t(X). t(g(h(X),h(Y))) :- q(X),q(Y). r(f(X)) :- t(X). s(h(f(X))) :- p(X). q(f(X)) :- p(X). p(a).	t(f(f(X))) :- t(X). t(g(h(a),h(a))) :- true. r(f(X)) :- t(X). s(h(f(a))) :- true. q(f(a)) :- true. p(a).
R^1	R^2

Fig. 1. Unfolding determinate predicates

Definition 11. determinacy-based widening

Let t_1 and t_2 be two predicates in an RUL program R, where t_1 depends on t_2. Let R^k be obtained from R as described above. Then apply widening to these two predicates if both t_1 and t_2 contain the same terms (modulo variable renaming) in their clause heads.

This widening is illustrated in the next section.

6 Example: String-Matching

These ideas are illustrated by applying them to the problem of specialising a naive string-matching procedure with respect to a given string but unknown text. The clauses of the procedure are as follows.

```
c0: match(P,T) <- m(P,T,P,T).

c1: m([],_,_,_) <- true.
c2: m([X|P],[X|T],P1,T1) <-
        m(P,T,P1,T1).
c3: m([X|_],[Y|_],P1,[_|T1]) <-
        X \= Y,
        m(P1,T1,P1,T1).
```

It is well-known that this program can be specialised with respect to a known ground first argument (such as `match([a,a,b],_)`) to yield an efficient matching procedure (like the Knuth-Morris-Pratt string matching procedure) that does not backtrack on the text string.

There are two important aspects of a successful specialisation of this program. Firstly, polyvariance is essential: the procedure **m/4** is replicated into several versions, one for each character in the string. Too much polyvariance is not necessarily bad but is inelegant. Too little polyvariance will not achieve the required result. Secondly, the right amount of unfolding is needed. Too little unfolding can yield a program that still backtracks on the text string when a mismatch occurs. Too much unfolding can give a program that "looks ahead" in the text string more than necessary.

Trace-terms can help to find the right amount of polyvariance and unfolding. The clause `c0` can be ignored for brevity, and we consider the goal `<- m([a,a,b],_,[a,a,b],_)`. There are obviously an infinite number of trace terms associated with successful computations of this goal. They can be enumerated (by some means not discussed here). The enumeration begins as shown in Figure 2, corresponding to the string being found in the text starting at the first four positions. The different trace-terms at a given position correspond to the different mismatches that can occur on the way to finding the match. (The character `*` stands for a character other than `a`. The character `@` stands for any character other than `a` and `b`. In the trace terms, a right square bracket is a "super-bracket" closing all unclosed left brackets).

Trace-term	Position where string occurs	text string
c2(c2(c2(c1]	1	[a,a,b,...]
c3(c2(c2(c2(c1]	2	[*,a,a,b,...]
c2(c2(c3(c2(c2(c2(c1]		[a,a,a,b,...]
c3(c3(c2(c2(c2(c1]	3	[*,*,a,a,b,...]
c3(c2(c2(c3(c2(c2(c2(c1]		[*,a,a,a,b,...]
c2(c3(c3(c2(c2(c2(c1]		[a,*,a,a,b,...]
c2(c2(c3(c2(c2(c3(c2(c2(c2(c1]		[a,a,a,a,b,...]
c3(c3(c3(c2(c2(c2(c1]	4	[*,*,*,a,a,b,...]
c3(c3(c2(c2(c3(c2(c2(c2(c1]		[*,*,a,a,a,b,...]
c3(c2(c3(c3(c2(c2(c2(c1]		[*,a,*,a,a,b,...]
c3(c2(c2(c3(c2(c2(c3(c2(c2(c2(c1]		[*,a,a,a,a,b,...]
c2(c3(c3(c3(c2(c2(c2(c1]		[a,*,*,a,a,b,...]
c2(c3(c3(c2(c2(c3(c2(c2(c2(c1]		[a,*,a,a,a,b,...]
c2(c2(c3(c2(c3(c3(c2(c2(c2(c1]		[a,a,@,a,a,b,...]
c2(c2(c3(c2(c2(c3(c2(c2(c3(c2(c2(c2(c1]		[a,a,a,a,a,b,...]

Fig. 2. Trace-terms for the Match procedure

A program describing this set of trace-terms is given in Figure 3. Determinate predicates in program are unfolded as explained in Section 5.2.

Here `t0` depends on `t2` and both have identical clause heads, so we create a recursive definition merging these two predicates (and renaming `t2` as `t0`). The result is in Figure 4. More answers are needed, beyond those tabulated in Figure 2, in order to obtain further widening. After more answers have been derived, the procedures for `t3` and `t8` will be merged, since they will both have the same terms in their clause heads. Notice that although `t7` also depends on `t3`, it will not be merged with it since its clause heads remain different. In fact `t1` will be merged with `t7` and `t0` will be merged with `t6`.

```
t0(c2(X)) :- t1(X).             t13(c2(c1)).
t0(c3(X)) :- t2(X).             t13(c3(c2(c2(c2(c1))))).

t1(c2(X)) :- t3(X).             t3(c2(c1))).
t1(c3(c3(X))) :- t6(X).         t3(c3(c2(X))) :- t7(X).

t6(c2(c2(X)) :- t10(X).         t5(c2(c2(X)) :- t12(X).
t6(c3(c2(c2(c2(c1))))).         t5(c3(c2(c2(c2(c1))))).

t10(c2(c1)).                    t7(c2(X)) :- t8(X).
t10(c3(c2(c2(c2(c1))))).        t7(c3(c3(c2(c2(c2(c1)))))).

t2(c2(X)) :- t4(X).             t8(c2(c1)).
t2(c3(X)) :- t5(X).             t8(c3(c2(c2(X))) :- t9(X).

t4(c2(X)) :- t11(X).            t9(c2(c1)).
t4(c3(c3(c2(c2(c2(c1)))))).     t9(c3(c2(c2(c2(c1))))).

t11(c2(c1)).                    t12(c2(c1)).
t11(c3(c2(c2(X))) :- t13(X).    t12(c3(c2(c2(c2(c1))))).
```

Fig. 3. Program Describing Trace-Terms for Match Program (after unfolding determinate predicates)

The final description of trace terms is in Figure 5. The three predicates in the program essentially determine three distinct versions of the `m/4` procedure. They are worth distinguishing, from the point of view of partial evaluation, since they behave differently in the computation. Of course, this is an approximation of the set of trace-terms actually derived; every one of the trace-terms tabulated above is described by this program.

Stability occurs when the set of trace-terms represents a complete set of answers. This can be determined by iterative fixed point techniques and is beyond the scope of this presentation.

A specialised program can be built directly from the regular program describing the trace-terms. Three version of `m/4` are generated, renamed appropriately. The trace-terms are used to decide how to unfold the clause bodies for each version.

```
match(X,T) :- m1(X,T,X,T).

m1([X|P],[X|T],P1,T1) :- m2(P,T,P1,T1).
m1([X|_],[Y|_],P1,[_|T1]) :- X \== Y, m1(P1,T1,P1,T1).

m2([X|P],[X|T],P1,T1) :- m3(P,T,P1,T1).
m2([X|_],[Y|_],[U|P1],[_,W|T1]) :- X \== Y, U \== W,
t0(c2(X)) :- t1(X).         t3(c2(c1))).
t0(c3(X)) :- t0(X).         t3(c3(c2(X))) :- t7(X).

t1(c2(X)) :- t3(X).         t7(c2(X)) :- t8(X).
t1(c3(c3(X))) :- t6(X).     t7(c3(c3(c2(c2(c2(c1)))))).

t6(c2(c2(X)) :- t10(X).     t8(c2(c1)).
t6(c3(c2(c2(c2(c1))))).     t8(c3(c2(c2(X))) :- t9(X).

t10(c2(c1)).                t9(c2(c1)).
t10(c3(c2(c2(c2(c1))))).    t9(c3(c2(c2(c2(c1))))).
```

Fig. 4. Regular Program after Widening `t0` and `t2`

```
t0(c2(X)) :- t1(X).        t3(c2(c1))).
t0(c3(X)) :- t0(X).        t3(c3(c2(X))) :- t1(X).

t1(c2(X)) :- t3(X).
t1(c3(c3(X))) :- t0(X).
```

Fig. 5. Final Description of Trace-Terms

```
        m1([U|P1],T1,[U|P1],T1).

m3([X],[X|_],_,_).
m3([X|_],[Y|_],[U|P1],[_,U|T1]) :- X \== Y,
        m2(P1,T1,[U|P1],[U|T1]).
```

This program is in fact more general than needed for matching the pattern `[a,a,b]`. In fact, it handles any pattern of the form `[X,X,Y]` where `X` is different from `Y`. A version specialised to `[a,a,b]` could easily be obtained from the program above by constant propagation.

7 Associating Trace-Terms with Process Trees in Functional Programming

We now consider an alternative way to use trace-terms to give control information. In Section 3.1 we showed how to associate a trace-term to a labelled computation. For non-ground terms, say *last (A:B:xs)*, it is possible to generate a labelled *process tree* representating a (possibly infinite) set of labelled computations. The notion of a process tree has been introduced for program specialisation in [SG95b]. Since process trees may be infinite, we generalise the labelled process tree to a finite tree, and finally extract the labels from the tree to obtain a recursively-defined set of trace-terms corresponding to the computations in the process tree.

Definition 12. labelled process tree

A *labelled process tree* T' is a (possibly infinite) process tree T such that for a node s in T:

- if s has edges leading to nodes $t_1, \ldots t_n$, $n \geq 1$, then there is a corresponding node s' in T' where s' is s annotated with the set of statement identifiers $\{\varphi_1/a_1, \ldots, \varphi_n/a_n\}$ where φ_i/a_i identifies the statement (or branch of the conditional) used to reduce s to t_i, $1 \leq i \leq n$.
- if s is the last node of a branch of T, there is a corresponding node s' in T' where s' is s annotated with the empty set to indicate the empty statement identifier.

For example, the labelled process tree for *last (A:B:xs)* is shown in Figure 6. For the following examples, we have used a simple generalisation rule for obtaining partial process trees from process trees: create a loop back to a previous node in a branch of the process tree if the two nodes are syntactially equivalent (modulo renaming). This generalisation rule does not ensure a finite partial process tree, but more powerful generalisation algorithms [SG95b] can be substituted for this simple generalisation rule.

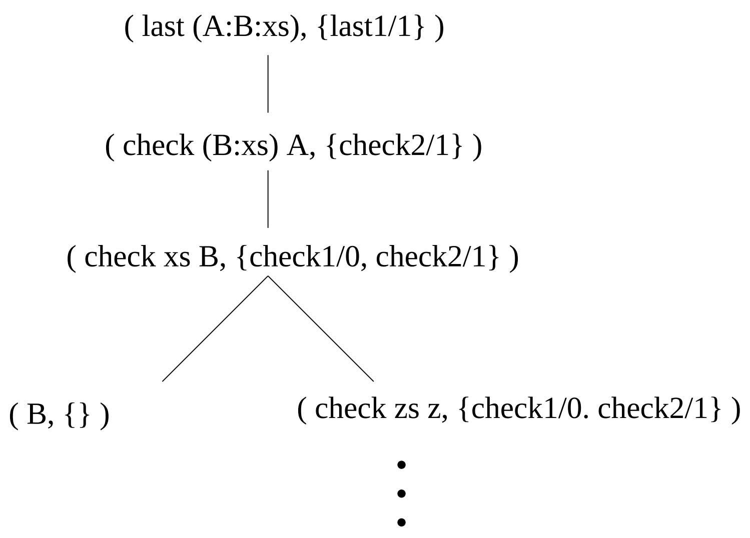
(last (A:B:xs), {last1/1})
(check (B:xs) A, {check2/1})
(check xs B, {check1/0, check2/1})
(B, {})
(check zs z, {check1/0. check2/1})

Fig. 6. The labelled process tree for *last (A:B:xs)*

Finally, we extract the trace-term tree from the labelled partial process tree in the following manner.

Definition 13. trace-term tree

A *trace-term tree* V is the tree obtained from a labelled partial process tree T such that for a node s in T

- if s is annotated with the set $\{\varphi_1/a_1, \ldots, \varphi_n/a_n\}$, then the corresponding node s' in V has n edges such that edge i is labelled with φ_i.
- if s is annotated with the set $\{\}$ and there is no edge leading from s to an ancestor of s in T, there is a node s' which is a leaf node.
- if s is annotated with the set $\{\}$ and there is an edge leading from s to t, an ancestor of s in T (indicating a loop), there is a corresponding edge in V leading from the corresponding node s' to t', the corresponding ancestor node in V.

The labelled partial process tree and trace-term tree for *last (A:B:xs)* are shown in Figure 7.

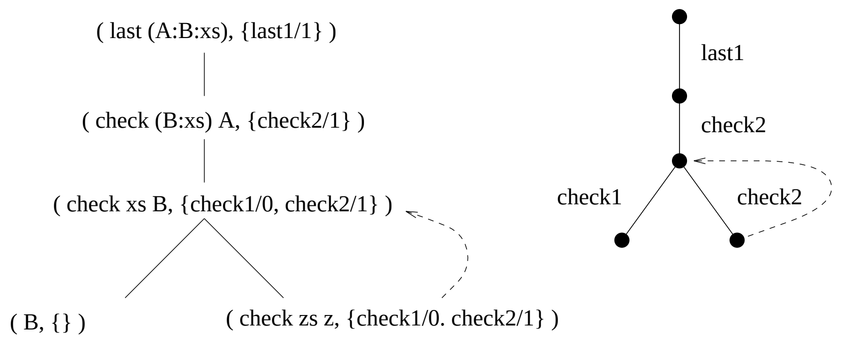
(last (A:B:xs), {last1/1})
(check (B:xs) A, {check2/1})
(check xs B, {check1/0, check2/1})
(B, {})
(check zs z, {check1/0. check2/1})
last1
check2
check1
check2

Fig. 7. The labelled partial process tree and trace-term tree for the term *last (A:B:xs)*

7.1 Pattern Matching Example

The pattern-matching example of Section 6 is now discussed in the context of functional programming. A process tree will be constructed, and descriptions of trace-terms extracted from it. The general tail-recursive pattern matching program p, written in the syntax of the language $\mathcal{M}$:

$match\, p\, s$	$loop\, p\, s\, p\, s$	$< m1/1 >$
$loop\, []\, ss\, op\, os$	$\rightarrow True$	$< lo1/0 >$
$loop\, (p : pp)\, []\, op\, os$	$\rightarrow False$	$< lo2/0 >$
$loop\, (p : pp)\, (s : ss)\, op\, os$	$\rightarrow$ **if** $p = s$ **then** $loop\, pp\, ss\, op\, os$ **else** $next\, op\, os$	
$next\, op\, []$	$\rightarrow False$	$< n1/0 >$
$next\, op\, (s : ss)$	$\rightarrow loop\, op\, ss\, op\, ss$	$< n2/1 >$

Assign the constructors $m1/1$, $lo1/0$, $lo2/0$, $lo3^T/1$, $lo3^F/1$, $n1/0$, $n2/1$, where $lo3^T/1$ represents the statement

$$loop\, (p : pp)\, (p : ss)\, op\, os \rightarrow loop\, pp\, ss\, op\, os$$

and $lo3^F/1$ represents the corresponding statement for the **else** branch. Then, the trace term for $match\,A{:}A{:}B\ \ A{:}A{:}A{:}A{:}B{:}u$ is:

$$m1(lo3^T(lo3^T(lo3^F(n2(lo3^T(lo3^T(lo3^F(n2(lo3^T(lo3^T(lo3^T(lo1]$$

Similarly, the trace term for the ground term $match\,A{:}A{:}B\,A{:}A{:}A$ is a single term.

$$m1(lo3^T(lo3^T(lo3^F(lo3^T(lo3^T(lo2]$$

As we noted for logic programs, enumerating these trace terms for successful computations in a program p will result in a set of trace terms from which the non-determinism in p can be identified.

Alternatively, we can construct the labelled process tree (Figure 8) and the corresponding trace-term tree (Figure 9) for the non-ground term $match\,A{:}\ A{:}\ Bu$.

Each of the choice points (circled nodes) of the trace-term tree indicates potential polyvariance in the final specialised program. Because positive driving was used to contruct the process tree, the full specialisation needed to achieve the Knuth-Morris-Pratt style pattern matcher is not achieved. Applying the widening technique discussed in Section 5.2 would merge the two uppermost circled nodes in the trace-term tree. However, with negative driving, the arc marked with an asterisk would be eliminated, and the required polyvariance is achieved. As a final footnote, there is slightly more polyvariance in the functional version than in the logic version, since the functional version deals explicitly with match failure. In the logic version, failure to find a match just means that the computation fails.

8 Using Trace-Terms to Detect Control Dependencies

Another use of trace-terms is to help decide when to unfold non-determinate atoms in logic programs. Unfolding too much non-determinism can cause an great increase in program size, with no computational benefits. On the other hand, sometimes unfolding non-deterministic atoms is the key to achieving the desired specialisation. If we are to unfold a non-deterministic choice, therefore, we want some indication that some useful specialisation results.

A simple example is given by the standard *solve* interpreter.

```
solve(true).
solve((x,y)) <- solve(x), solve(y).
solve(x) <- clause(x,y), solve(y).
```

An object program is represented, as usual, as a set of facts of form `clause(h,b)` where `h` is the clause head and `b` is the body. A simple propositional example is as follows:

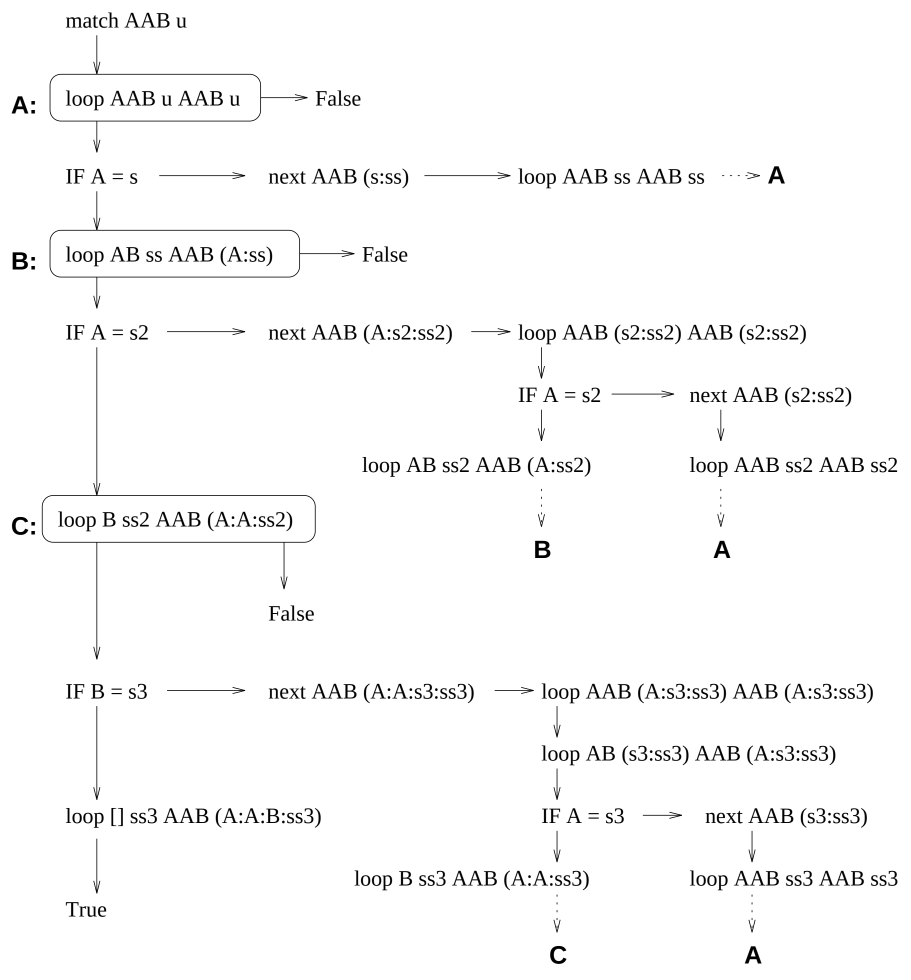

Fig. 8. The labelled partial process tree.

```
clause(r, true).
clause(r, (r,a)).
clause(a, true).
clause(a, a).
```

Partially evaluating with respect to `solve(r)` the obvious required specialisation is to unfold the calls to `clause(x,y)`. The desired specialised program is

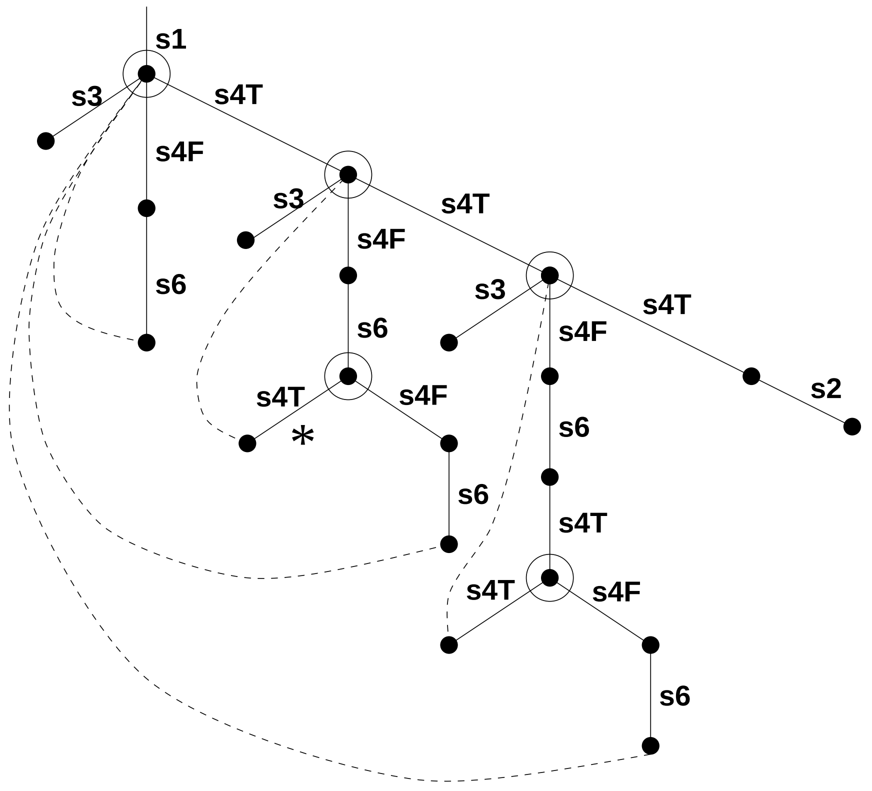

Fig. 9. The corresponding trace-term tree.

```
solve(r).
solve(r) <- solve(r), solve(a).
solve(a).
solve(a) <- solve(a).
```

The predicate `clause` is unfolded and eliminated from the specialised program. How can we detect that it is useful in this case to unfold the calls to `clause(x,y)`, which are non-deterministic?

Add trace-terms to the program, defining function symbols $s1/0$, $s2/2$, $s3/2$ representing the three `solve` clauses, and $c1/0$, $c2/0$, $c3/0$ and $c4/0$ the four `clause` facts. Answers derived using the third `clause` will have trace-terms of the form $s3(u, v)$, where u is one of the `clause` identifiers and v is one of the `solve` identifiers. Examining these, we will see that trace-terms are of one of the following forms: $s3(c1, s1)$, $s3(c2, s2(y, z))$, $s3(c3, s1)$ or $s3(c4, s3(w, z))$. In other words, we can see that the choice for `clause` (the first argument of $s3$) determines the choice for `solve` (the second argument). This indicates that unfolding `clause` will allow further specialising of the call to `solve`. If `clause(x,y)` were not unfolded in the third clause, then the call to `solve(y)` following it could not be specialised, since we can see from the trace-terms that all three clauses for `solve` are applicable to it.

9 Discussion

The idea of trace-terms has been defined in the context of logic and functional programming. Safe approximation of the set of trace-terms associated with a set of computations can be used to determine polyvariance and assist in the control of unfolding.

As noted above, trace-terms are similar in some ways to characteristic trees and histories. The main difference is that trace-terms relate to the *complete answers* to a goal, whereas chararacteristic trees and histories are based on some partial finite number of computation steps. In theory, therefore, trace-terms should give more precise information. However, more refined widening operators than the one presented here may need to be developed to take advantage of this. A second difference is that trace-terms do not depend on the computation rule. Examination of trace-terms can yield information such as dependencies between subgoals in a clause body (or subterms in a definition), as sketched in Section 8.

The practicalities of using trace-terms have not been fully discussed in this paper. One important question which has to be studied is: how should trace-terms be enumerated? There are different possibilities. The most likely ones seem to be goal-oriented fixpoint methods like OLDT in logic programming [TS86] and minimal function graphs in functional programming [JM86]. Complexity issues have not been studied carefully yet either.

The widening operator discussed in Section 5.2 is effectively a generalisation operator, since it introduces recursive descriptions of trace-terms. There appear to be useful possibilities in combining trace-term generalisation with other methods based on termination analysis. This ideas was implicit in the treatment of the functional pattern-match program, in which as finite process tree was first contructed. An example of related structures in logic programming are m-trees [MG95] The methods described by Leuschel and Martens [LM96] make use of characteristic trees in a generalisation technique controlled by *homeomorphic embeddings*. It would be interesting to study the behaviour of trace-terms instead of characteristic trees in this approach. Trace-terms have a different structure to characteristic trees and are like to give different embeddings.

Control based on termination analysis, generally speaking, produces the greatest possible unfolding of a computation. Recursive descriptions of sets of trace-terms can be extracted from structures built based on termination analysis, such as m-trees and process trees, and then widening can be applied to decide on suitable polyvariance. This idea might be useful where m-trees or process-graphs, though finite, contain “too much” polyvariance; trace-terms can be used to capture the "useful" polyvariance.

Note that extracting trace-terms from (branches of) such structures as m-trees and process-trees is essentially no different than getting them from individual concrete computations. The only information required is the identification of statements with arcs of the graphs or trees. Backward arcs in these structures translate directly to recursive descriptions of set sof trace-terms.

Comparison of the application of these techniques to logic and functional programs should yield some interesting comparisons. Capturing determinacy has been a concern of logic program specialisation for some time. In the functional setting, control flow information yielded by trace-terms may provide more flexible control than that given by binding-time analyses, since "static" seems to correspond roughly to "determinate" in functional languages (though not in relational languages). Furthermore, cases where determinacy is obtained with partially static data should also emerge. In new integrated functional and logic languages such as Escher [Llo95] which allow computation with partly instantiated terms, an analysis of control covering both logic and functional features is desirable.

Acknowledgements

We would like to thank the referees for helpful comments, and the participants at the Dagstuhl Seminar on Partial Evaluation for stimulating discussions.

References

[BJ92] M. Bruynooghe and G. Janssens. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.

[dWG94] D.A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12), Nancy*, 1994.

[FSVY91] T. Frühwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings of the IEEE Symposium on Logic in Computer Science, Amsterdam*, July 1991.

[Gal86] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI-86), Brighton*, pages 109–122, 1986.

[GB91] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[GdW94] J. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*, MIT Press, 1994.

[GK93] Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In G. Filè P.Cousot, M.Falaschi and A. Rauzy, editors, *Static Analysis. Proceedings*, pages 112–123, Springer-Verlag, 1993.

[GS94] Robert Glück and Morten Heine Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, pages 165–181, Springer-Verlag, 1994.

[HCCar] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, (to appear).

[Hei92] N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Symposium and Conference on Logic Programming*, pages 765–769, MIT Press, 1992.

[JM86] N. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proceedings of Principle of Programming Languages (POPL'86)*, ACM Press, 1986.

[Jon87] N. Jones. Flow analysis of lazy higher order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Ellis-Horwood, 1987.

[Leu95] M. Leuschel. Ecological partial deduction: preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, Springer-Verlag (to appear), 1995.

[Llo95] J.W. Lloyd. *The Programming Language Escher.* Technical Report CSTR-95-013, Dept. of Computer Science, University of Bristol, 1995.

[LM96] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proc. of the Dagstuhl Seminar on Partial Evaluation*, Springer-Verlag, 1996.

[MG95] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proc. International Conference on Logic Progrmaming, (ICLP'95), Tokyo*, MIT Press, 1995.

[SG95a] H. Sağlam and J. Gallagher. *Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions.* Technical Report CSTR-95-17, University of Bristol, Department of Computer Science, 1995.

[SG95b] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *International Logic Programming Symposium*, page to appear, MIT Press, 1995.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP*, Springer-Verlag, 1994.

[Sha87] E.Y. Shapiro. Or-parallel prolog in flat concurrent prolog. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers (Volume 2)*, MIT Press, 1987.

[TS86] H. Tamaki and T. Sato. OLDT resolution with tabulation. In E.Y. Shapiro, editor, *Proc. 3rd ICLP, London*, Springer-Verlag, 1986.

[Tur88] V. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation*, pages 531–549, North-Holland, 1988.

[Tur93] V. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[Tur96] V. Turchin. Metacomputation: MST plus SCP. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proc. of the Dagstuhl Seminar on Partial Evaluation*, Springer-Verlag, 1996.

[YS90] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.