

A System For Specialising Logic Programs¹

J.P. Gallagher

November 1991

Department of Computer Science
University of Bristol
Queen's Building
University Walk
Bristol BS8 1TR
U.K.

e-mail: john@compsci.bristol.ac.uk

¹Work partly supported by ESPRIT Project PRINCE (5246)

Abstract

This report describes SP, a system for specialising logic programs. The report functions as a user's manual for SP, and also contains the algorithms employed and arguments for their correctness. A number of examples of program specialisation are given in Appendix A.

Contents

1	Program Specialisation	4
2	Transformations in SP	5
3	Unfolding Rules	10
4	Approximation	13
5	The Specialisation Algorithm	16
6	How to Use SP	20
7	Discussion	23
A	Examples of Specialisation	28
B	Unfoldability Conditions for Built-ins	36

1 Program Specialisation

SP is a system for specialising logic programs. Before describing the system, it is worth reviewing briefly the aims and interesting applications of program specialisation.

To specialise a program is to restrict its behaviour in some way. The purpose of specialisation is to exploit the restriction to gain efficiency. A specialised program is equivalent, within the bounds of the restriction imposed, to the original unspecialised program, but should be more efficient. The uses of specialisation arise because it is sometimes convenient or concise to define a program as a special case, or restriction, of a given more general program.

1.1 The Specialisation Principle in Computing

Specialisation is a simple but important principle with applications in basic areas of computing. The elegant reconstruction of the theory of compilation using specialisation is an indication of its significance. Futamura, inspired by ideas of Lombardi [21] was apparently the first explicitly to present compilation as specialisation, or “partial evaluation” of an interpreter [5], and Ershov [4] later independently developed the related idea of “mixed computation” of an interpreter. Both authors developed definitions of compiled code, compilers and compiler-generators; these definitions are sometimes called the Futamura Projections.

The first Futamura projection, defining compilation, can be briefly explained as follows. An interpreter for some language L is a meta program taking a program in L as input data; the result of restricting the interpreter to a particular object program P_0 is a specialised interpreter that can interpret only one program, P_0 . This specialised interpreter may be considered as a compiled version of P_0 ; more specifically, it is the result of compiling P_0 into the language in which the interpreter is written. The definitions of compilers and compiler-generators are based on self-application of a program specialiser. Such ideas have been put into practice, most notably by Jones *et al.* [14].

Extending the idea of compilation, any program P_1 in language L_1 can be translated into a program in language L_2 ; this is done by giving operational semantics for L_1 in the form of an interpreter written in L_2 , and then specialising the interpreter for P_1 . The result “compiles” P_1 using the semantics specified by the interpreter. This translation could be from a higher to a lower-level language, or vice versa, or be within the same language.

The translation could serve to compile programs using non-standard operational semantics, by specialising an interpreter defining the required semantics. To name a few possibilities, meta programs could be used to specify interpreters for virtual machines, abstract interpretations [23], language extensions [24] and parallel or non-standard execution strategies [8] [2].

1.2 Future Applications

More speculatively, a further range of applications is suggested by regarding proof procedures for various formal logics as meta programs. Usually in such cases - take first order logic as an example - one does not readily regard the proof procedure as a language interpreter. The fact that this is only a psychological barrier was shown by the practicality of logic programming. Here a proof procedure for a subset of first order logic is sufficiently efficient to be regarded as a program interpreter. But a proof procedure for the whole of first order logic, written

in a suitable meta language, has equal claim to be a language interpreter, albeit not very practical, giving procedural semantics to any first order theory.

The specialisation of the proof procedure for a fixed theory T in first order logic may provide a means to compile T into a more obviously computational form. This idea has a superficial connection to the ideas of the school of program synthesis that regards programs as proofs in a constructive logic. The program that is synthesised from a proof using that method can perhaps be seen as a specialised version of the constructive proof procedure, specialised to perform only one proof.

There is a possibility of using specialisation to compile very high-level languages such as constraint logic programming languages. A number of other challenging problems and applications for program specialisation in software development and artificial intelligence are discussed in [13].

2 Transformations in SP

2.1 The Aims of the SP System

The SP system is designed to specialise declarative logic programs. It does not aim to handle “full Prolog” as Mixtus [25], for example, does. However declarative logic programs may include many of the built-in predicates normally implemented in Prolog, such as those handling arithmetic. Such predicates are declarative since they can in principle be given (possibly infinite) definitions as logic programs. A list of the allowed Prolog built-ins is given in Appendix B.

The SP system itself is not written completely declaratively, and hence SP is not self-applicable. The main intention is for SP to be a general purpose specialiser rather than a compiler production kit, hence self-application is not a primary goal. In any case, many of the more interesting applications of specialisation do not involve self-application. However, a longer term goal is to rewrite SP in a declarative language and experiment with self-application.

2.2 Specialisation of Logic Programs

The aim of program specialisation in logic programming is to take a program P and a goal G , and derive a program P' with the following properties:

1. Computations of $P \cup \{G\}$ and $P' \cup \{G\}$ give identical results (computed answers and finite failures).
2. Computation of $P' \cup \{G\}$ is more efficient than $P \cup \{G\}$.

Although specialisation by partial evaluation is conceptually a simple transformation, the main problems of achieving effective program specialisation are:

- Control of unfolding: too little unfolding may not produce the required specialisation; too much unfolding can cause exponential growth in program size; the wrong unfoldings can actually decrease the efficiency of specialised programs [32].

- Termination: partial evaluation of a program may give an infinite computation tree, and the partial evaluation procedure may generate an infinite number of finite trees.
- Correctness: with one or two notable exceptions [20], [1], the correctness of previous partial evaluation methods has not been formally justified.

The following techniques, incorporated in SP have been developed largely in response to these problems.

- Control of unfolding: determinacy in the computation is the basis for the control of unfolding. This gives better control than other methods. More specific clause transformations are introduced to obtain specialisation without unfolding.
- Termination: the flow analysis algorithm incorporates an abstraction operation that can be used to ensure termination.
- Correctness: the correctness of each transformation is justified.

In the next section definitions are given of partial evaluation and other operations used in SP. SP is called a specialisation system rather than a partial evaluation system, though SP relies mostly on partial evaluation. Partial evaluation in logic programming has also been called “partial deduction” in [16]; both terms are strongly operational and associated with computation trees.

The terms “specialisation” and “partial evaluation/deduction” are sometimes used synonymously, though specialisation includes partial evaluation as well as other transformations. For instance the notion of a “more specific logic program” [22] is a kind of specialisation used in SP, in which a logic program clause may be replaced by a more instantiated version, or even eliminated from the program, without altering the program’s success or finite failure set. Transformations such as folding and renaming may also be incorporated in the specialisation process.

Furthermore the use of abstract interpretation and other safe approximations of program behaviour can give specialisations that would require “infinite” partial evaluation.

2.3 Definitions

For logic programming, the terminology defined in [19] is used. The next four definitions concerning partial evaluation are quoted from [20].

Definition 2.1 *resultant*

A resultant is a first order formula $Q_1 \leftarrow Q_2$ where Q_i is either absent or a conjunction of literals, and any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Definition 2.2 *resultant of a derivation*

Let P be a normal program and $G_0 = \leftarrow G$ be a normal goal. Let G_0, \dots, G_n be an SLDNF derivation of $P \cup \{G_0\}$, where $G_n = \leftarrow Q$ and $\theta_1, \dots, \theta_n$ is the sequence of substitutions associated with the derivation. Let $\theta = \theta_1\theta_2 \dots \theta_n$. Then the derivation has length n , computed answer $\theta|_G$ and resultant $G\theta \leftarrow Q$. Note that if G is an atom then the resultant is a normal clause.

Definition 2.3 *partial evaluation*

Let P be a normal program and A an atom. Let T be an SLDNF tree for $P \cup \{\leftarrow A\}$, and let $\leftarrow G_1, \dots, \leftarrow G_n$ be goals chosen from the non-root nodes of T such that there is exactly one goal from each non-failing branch of T . Let $\theta_1, \dots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \dots, \leftarrow G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1, \dots, A\theta_n \leftarrow G_n\}$ is called a partial evaluation of A in P .

If \mathbf{A} is a finite set of atoms, then a partial evaluation of \mathbf{A} in P is the union of the partial evaluations of the elements of \mathbf{A} .

Definition 2.4 *closed*

Let S be a set of first order formulas and \mathbf{A} a finite set of atoms. Then S is \mathbf{A} -closed if each atom in S containing a predicate symbol occurring in \mathbf{A} is an instance of an atom in \mathbf{A} .

Given an atom A and a program P , there exist in general infinitely many different partial evaluations of A in P . Some fixed rule for generating resultants is used, called an *unfolding rule*.

Definition 2.5 *unfolding rule*

An unfolding rule U is a function which given P and A , returns exactly one finite set of resultants that is a partial evaluation of A in P . If \mathbf{A} is a finite set of atoms and P a program, then the set of resultants obtained by applying U to each atom in \mathbf{A} is called a partial evaluation of \mathbf{A} in P using U .

Definition 2.6 An operation $\text{abstract}(S)$ is any operation satisfying the following condition. Let S be a set of atoms; then $\text{abstract}(S)$ is a set of atoms A with the same predicates as those in S , such that every atom in S is an instance of an atom in A .

A constructive definition of abstract can be given by defining an *approximation* function on atoms, called α , such that if A is an atom, $\alpha(A)$ is an atom and A is an instance of $\alpha(A)$. The atom $\alpha(A)$ is called the *representative* of A . $\text{abstract}(S)$ is then defined as $\{ \alpha(A) \mid A \in S \}$.

Definition 2.7 *renaming definition*

Let A be an atom, and let x_1, \dots, x_n be the distinct variables in A , in order of their first occurrence. Let p be a predicate symbol different from that in A . Then the clause $p(x_1, \dots, x_n) \leftarrow A$ is called a renaming definition for A .

Let \mathbf{A} be a set of atoms; then a set of renaming definitions for \mathbf{A} is the set of renaming definitions for the atoms in \mathbf{A} , with a distinct predicate name chosen for each definition.

Example 1 Let $\mathbf{A} = \{P(F(x), y), P([], x)\}$. Then suitable renaming definitions for \mathbf{A} are:

$$P1(x, y) \leftarrow P(F(x), y)$$

$$P2(x) \leftarrow P([], x)$$

Definition 2.8 *unfold*

Let P be a normal program and let $C = A \leftarrow Q$ be a clause in P . Then a clause $A\theta \leftarrow R$ is obtained by unfolding C in P if there is an SLDNF derivation of $P \cup \{\leftarrow Q\}$ of the form $\leftarrow Q, \dots, \leftarrow R$, with computed answer θ , and the clause C is not used in the derivation (or any sub-derivation) except possibly at the first step.

The program $P' = P - \{C\} \cup \{C'\}$ is obtained from P by unfolding C in P .

NOTE. The restriction on the use of C in the derivation is for compatibility with the usual definition of unfolding (e.g. [31], [28]), where unfolding is a single derivation step. Hence the clause C is not available after the first derivation step.

Definition 2.9 *(complete) renaming transformation*

Let P be a normal program, and $C = A \leftarrow Q_1, B\theta, Q_2$ be a clause in P , where $B\theta$ is a literal, and Q_i are conjunctions of literals. Let R be a set of renaming definitions whose head predicates do not occur in P .

- Suppose B is an atom; let $D = B' \leftarrow B$ be in R . Then let clause $C' = A \leftarrow Q_1, B'\theta, Q_2$.
- Suppose B is a literal $\neg N$; let $D = N' \leftarrow N$ be in R . Then let clause $C' = A \leftarrow Q_1, \neg N'\theta, Q_2$.

In both cases, C' is obtained from C by renaming $B\theta$ with the renaming clause D .

A complete renaming of a clause (wrt to a set of renaming definitions) is obtained by renaming every atom in the clause body for which a renaming definition exists. Note that the result is not uniquely defined; an atom may be renamed by more than one definition.

Lemma 2.10 Let P be a program and let A be an atom. Let P_A be a partial evaluation of A in P . Let $A' \leftarrow A$ be a renaming definition for A , where the renaming predicate in the head does not occur in P . Then for each resultant $A\theta \leftarrow R$ in P_A , a clause $A'\theta \leftarrow R$ can be obtained by unfolding the clause $A' \leftarrow A$ in the program $P \cup \{A' \leftarrow A\}$.

Definition 2.11 *most specific generalisation*

Let T be a non-empty set of terms. A generalisation of T is a term t such that $\forall s \in T . s$ is an instance of t . t is a most specific generalisation of T if for all t' such that t' is a generalisation of T , t is an instance of t' . t is unique modulo variable renaming, so a most specific generalisation of T is referred to as $msg(T)$.

The concept of more specific programs was introduced in [22]. The same ideas are constructed below in a somewhat different way.

Definition 2.12 *more specific goal*

Let P be a normal program and G a goal. Define D_k , $k = 1, 2, \dots$ to be the set of resultants of SLDNF derivations of length k of $P \cup \{\leftarrow G\}$ with some fixed safe computation rule. If $D_k \neq \emptyset$, define G' as:

$$G' = msg \left\{ G\theta \mid G\theta \leftarrow R \in D_k \right\}$$

Assuming G' is chosen so that it has no variable in common with G , an mgu of G and G' exists, say ϕ_k . $G\phi_k$ is called a more specific version of G .

Definition 2.13 *more specific clause*

Let P be a normal program and $C = H \leftarrow B$ a clause in P . Let $k > 0$ and let $\leftarrow B\phi_k$ be a more specific version of $\leftarrow B$ (if it exists). Define C_k to be $H\phi_k \leftarrow B\phi_k$. C_k is called a more specific version of C .

Definition 2.14 *more specific program*

Let P be a normal program, and let $k > 0$. Let P' be obtained by replacing some clause C in P by a more specific version C_k , if it exists, or deleting C if C_k does not exist. Then P' is called a more specific version of P .

Lemma 2.15 *Let P be a normal program, and let P' be a more specific version of P . Then for all goals G , $P \cup \{G\}$ succeeds with computed answer θ (resp. fails finitely) iff $P' \cup \{G\}$ succeeds with computed answer θ (resp. fails finitely).*

PROOF. Omitted. □

The definition of more specific program transformations is designed to preserve SLDNF procedural semantics. Note that in some circumstances even more specific programs can be found by detecting and eliminating infinite failed derivations (e.g. by abstract interpretation). Such transformation could of course change the SLDNF procedural semantics of the program (by turning infinite loops into finite failures) but it could be useful in some cases [3]. Such transformations are not used in SP, but will be incorporated in later versions. For the moment a modified definition of Definition 2.13 is as follows.

Definition 2.16 *non-failing more specific programs*

A derivation of length k is non-failing if it is the prefix of at least one successful derivation.

Let P be a normal program and $C = H \leftarrow B$ a clause in P . Define D_k , $k = 1, 2, \dots$ to be the set of resultants of non-failing SLDNF derivations of length k of $P \cup \{\leftarrow B\}$ with some fixed safe computation rule. If $D_k \neq \emptyset$, define B' as:

$$B' = msg \left\{ B\theta \mid B\theta \leftarrow R \in D_k \right\}$$

An mgu of B and B' exists, say ϕ_k . Define C_k to be $H\phi_k \leftarrow B\phi_k$. C_k is called a non-failing more specific version of C .

A non-failing more specific version of a program P is defined similarly as in Definition 2.14.

It is undecidable whether a derivation is non-failing, so in practice only a superset of D_k can be computed, by using sufficient conditions for detecting failing derivations.

Definition 2.17 *imported*

When specialising a program, certain predicates are designated imported. These are predicates defined in another program module. Literals containing these predicates cannot be selected during any SLDNF derivation in the specialisation procedures.

Definition 2.18 *built-in predicates*

Certain Prolog built-in predicates may appear in programs to be specialised. The built-ins that are allowed are predicates that in principle are declarative and could be defined with a set of normal clauses. (Others such as *assert*, *retract*, *var*, etc., have no such definitions.) For each acceptable built-in predicate p , there are conditions which must be satisfied before literals containing p are evaluated. The acceptable built-ins, and their evaluation conditions, are given as a Prolog program in Appendix B.

SLDNF derivations may include the execution of built-ins whose evaluation conditions are satisfied.

3 Unfolding Rules

Most partial evaluation systems treat the control of unfolding as primarily a loop prevention problem (e.g. [25]). In other words, it is seen as desirable to unfold as much as possible without risking unfolding an infinite branch of the computation tree.

In fact, loop prevention can be a rather poor basis to control partial evaluation. It is certainly not the case that it is a good idea to do as much unfolding as possible during partial evaluation. Good control decisions are as critical in partial evaluation as they are at run time. The main factors affecting unfolding in SP are determinacy and choice points in the computation tree. This approach tends to yield somewhat more conservative unfoldings than with loop prevention schemes, but in conjunction with more specific goal transformations gives better quality specialised program. The termination problem is seen as a problem in static analysis, not a control problem, and SP uses methods drawn from abstract interpretation (Section 4).

In this section, several variations of an unfolding rule are given. The basic rule uses *determinacy* to control the unfolding.

3.1 Determinate Unfolding

Definition 3.1 Let G be a normal goal and P a program. G is called *dead* if there is some atom (not having an imported predicate) in G that unifies with no clause head in P , or there is an evaluable built-in in G that fails. A goal is *live* if it is not dead.

Definition 3.2 Let G be a goal and P a program. G is called *determinate* if G is live and either:

- there is an atom A in G such that there is exactly one live resolvent G' obtainable by resolving G on A with a clause in P ; or,
- G contains a ground negative literal whose solution yields the resolvent G' ; or

- there is an evaluable built-in literal whose execution gives exactly one live resolvent G' .

The derivation of G' from G is called a *determinate resolution*.

Example 2 Let P be the program:

```
P(A) <-
P(B) <-
Q(B) <-
Q(C) <-
Q(D) <-
```

Let G be the goal $\leftarrow P(x), Q(x)$. G is determinate; by selecting $P(x)$, the resolvents $\leftarrow Q(A)$ and $\leftarrow Q(B)$ are obtained. Exactly one of these (the second) is live. Note that each of the atoms in G matches more than one clause head.

The basic unfolding rule in SP (called the *determinate rule*) is as follows.

Definition 3.3 *determinate unfolding rule*

Let A be an atom and P a program. Then the partial evaluation of A in P is the set of resultants $A\theta \leftarrow Q$, where:

1. there is an SLDNF derivation $\leftarrow A, \dots, \leftarrow Q$ and each goal in the derivation apart from (possibly) $\leftarrow A$ and $\leftarrow Q$ is determinate;
2. $\leftarrow Q$ is live and not determinate;
3. each derivation step apart from the first is a determinate resolution.

In other words this strategy unfolds at least one step, and then continues as far as the first goal that is not determinate. The only possible non-determinate step is the first.

3.2 Determinate Unfolding With Depth Bound

Determinate unfolding could lead to infinite unfolding in the unlikely case that there was a determinate computation of unbounded length, so a depth bound can be imposed. The solution of negative ground literals could also be infinite. To ensure termination in all cases, the unfolding rule *determinate unfolding with depth bound* is defined as follows:

Definition 3.4 *determinate unfolding with depth bound*

Let A be an atom and P a program. Let $k > 1$ be a positive integer (the depth bound). Then the partial evaluation of A in P is the set of resultants $A\theta \leftarrow Q$, where:

1. there is an SLDNF derivation $\leftarrow A, \dots, \leftarrow Q$ and each goal in the derivation apart from (possibly) $\leftarrow A$ and $\leftarrow Q$ is determinate; and

2. $\leftarrow Q$ is live; and
3. $\leftarrow Q$ is not determinate, or the length of the derivation up to $\leftarrow Q$ is k , or $\leftarrow Q$ is determinate and the SLDNF tree for a selected negative ground literal in $\leftarrow Q$ has height greater than k ; and
4. each derivation step apart from the first is a determinate resolution, where successful solutions of ground negative literals in the derivations have finitely failed SLDNF trees at most k in height.

This is just a modification of the determinate rule which stops unfolding as soon as the depth bound is reached - in other words the final goal $\leftarrow Q$ may also be determinate.

3.3 Determinacy With More Specific Transformations

More determinacy can sometimes be squeezed from a computation by interspersing more specific clause transformations with unfolding.

Example 3 *Let P be:*

```

c1:      P(F(A)) <-
c2:      P(F(B)) <-

c3:      Q(x) <- S(x)
c4:      Q(x) <- R(x)

c5:      S(F(A)) <-
c6:      R(G(C)) <-

c7:      T(x) <- P(x), Q(x)

```

The atom $T(z)$ has the single resultant $T(z) \leftarrow P(z), Q(z)$ using determinate unfolding. However, this resultant has a more specific version $T(F(w)) \leftarrow P(F(w)), Q(F(w))$ obtained by constructing derivations of length 1 and selecting $P(z)$. Now the body is determinate since resolving with c4 gives a dead goal, and so further determinate unfolding can be performed to give the resultant $T(F(A)) \leftarrow \text{true}$.

Definition 3.5 *more specific resolution*

Let P be a program and G a goal. Given any goal H , assume that there is some algorithm for computing a more specific version of H wrt P . Let G' be obtained from G by resolving with a program clause. Let $G'\phi$ be a more specific version of G' . Then $G'\phi$ is obtained from G by more specific resolution.

This idea can be incorporated in an unfolding rule as follows.

Definition 3.6 *determinate unfolding with more specific transformations*

The rules for determinate unfolding (both with and without depth bounds) are modified by performing more specific resolution steps in place of resolution steps.

Note that the implementation of this unfolding rule in SP does not follow the definition directly; more specific versions are not computed on each derivation step, but only when a non-determinate derivation step is reached.

3.4 Executable Goals

Some further control may be exercised over unfolding by designating certain atomic goals as *executable* even when they are not determinate. Such atomic goals should have a finite number of solutions.

Examples of atoms that are executable by default are `member(x, l)` where `l` is a list of known length, `append(x, y, l)`, where `l` is a list of known length, and the built-in `clause(x, y)` where `x` is non-variable. In all these cases there is a finite number of solutions. The user may declare other atoms executable; this is explained in Chapter 6.

Executable atoms can be executed if they appear at the left of the resolvent after all determinate choices have been made. An extension to the determinate unfolding rule is as follows:

Definition 3.7 *determinate unfolding with executables*

Let A be an atom and P a program. Let $A\theta \leftarrow B, Q$ be a resultant in the partial evaluation of A in P using the determinate unfolding rule (possibly using more specific resolvents). Suppose B is an executable atom, whose solutions are $\{\phi_1, \dots, \phi_n\}$. Then using determinate unfolding with executables, the resultants $\{A\theta\phi_1 \leftarrow Q\phi_1, \dots, A\theta\phi_n \leftarrow Q\phi_n\}$ replace $A\theta \leftarrow B, Q$ in the partial evaluation of A in P .

The user has responsibility to check whether an atom designated as executable has a finite number of solutions.

3.5 No Unfolding

For some programs, useful optimisations can be performed with no unfolding at all (see [10]). Such optimisations can be performed by the SP system. In this case, the unfolding rule is as follows:

Definition 3.8 Let A be an atom and P a program. Let $k > 1$ be a positive integer (the depth bound). Then the partial evaluation of A in P is the set of resultants $A\theta \leftarrow Q\theta$, where there is a clause $A' \leftarrow Q$ in P and an mgu of A and A' is θ .

4 Approximation

The operation *abstract* in the flow analysis algorithm is critical for the quality of specialisation. Crude methods such as *depth-k* abstractions on atoms [31] could be used to enforce termination

but the results are not generally satisfactory. In [11] the notion of *characteristic path* was introduced and used to construct an abstract interpretation. The method used in the SP system is based on the same ideas.

A characteristic path (or tree) can be thought of as giving the structure of an SLDNF derivation (tree). It records the clauses selected in the derivation (or tree), and the position of the literals resolved upon, abstracting the information about the resolvents. The intuitive idea behind the abstraction in SP is that if two atoms A and B have the same characteristic tree, then they are regarded as equivalent from the point of view of partial evaluation, and represented by a single atom.

Example 4 Let P be the usual program for `append(x,y,z)`.

Let G_1 be `<- append([1,2,3],[4,5],z)` and G_2 be `<- append([A,B,C],[D,E],z)`. The successful derivations of G_1 and G_2 have the same characteristic paths. The constants `1,2,A,B,...` and so on play no part in the derivations. The goals G_1 and G_2 are equivalent as regards their unfoldings.

Definition 4.1 *characteristic path*

Let P be a normal program and let G be a goal. Assume that the clauses in P are numbered. Let G, G_1, \dots, G_n be an SLDNF derivation of $P \cup \{G\}$. The characteristic path of the derivation is defined as the sequence $(l_1, c_1), \dots, (l_n, c_n)$, where l_i is the position of the selected literal in G_i , and c_i is defined as:

- if the selected literal is an atom, then c_i is number of the clause chosen to resolve with G_i ;
- if the selected literal is $\neg p(t)$, c_i is the atom p .

Definition 4.2 *characteristic tree*

Let A be an atom and P a program. Let U be an unfolding rule. Then the characteristic tree of the partial evaluation of A in P using U is the set of characteristic paths of the derivations used to form the partial evaluation.

Given two atoms with the same characteristic tree, which atom should then be chosen to represent them? The obvious idea would be to take their msg, but unfortunately the characteristic tree of the msg of atoms A and B is not in general the same as that of A and B , in particular using the determinate unfolding rule.

Example 5 Let P be:

```
1:  Eqlist([],[]) <-
2:  Eqlist([x|xs],[y|ys]) <- Eqlist(xs,ys).
```

Let A be `Eqlist([1,2],w)` and B be `Eqlist(w,[3,4])`. A and B have the respective partial evaluations, using the determinate unfolding rule:

$\text{Eqlist}([1,2], [1,2]) \leftarrow$

and

$\text{Eqlist}([3,4], [3,4]) \leftarrow$

The characteristic tree for both A and B is $\{((1,2), (1,2), (1,1))\}$. But $\text{msg}(\{A, B\}) = \text{Eqlist}(\mathbf{x}, \mathbf{x})$, which does not have the same partial evaluation as A and B , since the goal $\leftarrow \text{Eqlist}(\mathbf{x}, \mathbf{x})$ is not determinate.

The cause of this problem is that different arguments of an atom may independently force the same determinate choice during a derivation. The way around the problem is to take the msg of the heads of the clauses in the partial evaluations of the atoms. For the example, this is $\text{msg}(\{\text{Eqlist}([1,2], [1,2]), \text{Eqlist}([3,4], [3,4])\}) = \text{Eqlist}([x, y], [x, y])$. This has the same partial evaluation as A and B using the determinate unfolding rule. Note that neither A nor B is an instance of $\text{Eqlist}([x, y], [x, y])$.

The abstraction operation can now be stated precisely.

Definition 4.3 *abstraction using characteristic trees*

Let S be a set of atoms, and define S_c as the subset of S having characteristic tree c . For each atom A in S , let R_A be the partial evaluation of A in P . Define $\text{head}(A)$ as

$$\text{head}(A) = \text{msg} \left\{ A\theta \mid A\theta \leftarrow Q \in R_A \right\}.$$

Define S' as

$$S' = \left\{ \text{head}(A) \mid A \in S \right\}.$$

Now define $\text{abstract}(S)$ as

$$\bullet \text{ abstract}(S) = \left\{ \text{msg}(S'_c) \mid \begin{array}{l} c \text{ is the characteristic tree} \\ \text{of some } A \in S' \end{array} \right\}$$

4.1 Other Abstraction Mechanisms

The use of characteristic trees is a useful heuristic, but the development of other means of abstraction is one area in which SP is being improved.

The main shortcoming of partial evaluation, which has only partly been tackled in the SP system, is that contextual information in the computation tree is lost during partial evaluation. That is, assume that an atom A has a partial evaluation that includes a resultant $A\theta \leftarrow B_1, B_2$. The literals B_1 and B_2 are then themselves partially evaluated, but any dependence (shared variables) between the two is lost. One way of viewing the problem is to incorporate “bottom-up” computation into the partial evaluation, getting information about

the possible solutions to B_1 and B_2 . The use of more specific clause transformations is a step in this direction but better results can be achieved.

Incorporation of abstract substitutions into the specialisation process along the lines discussed in [9] is one possibility. Roughly, in the above example, an abstract answer β could be computed for the goal $\leftarrow B_1, B_2$, and then the atoms $B_1\beta$ and $B_2\beta$ would be partially evaluated using some abstract partial evaluation procedure. In this way every atom is evaluated in its (abstract) context.

Ideas for incorporating abstract substitutions into partial evaluation are discussed in [3].

5 The Specialisation Algorithm

The SP system takes as input a normal program P and a goal G . The output is a specialised program, which may be written to a file for later execution. The algorithm is in three parts:

1. Flow analysis.
2. Construction of the Specialised Program.
3. Cleaning up operations.

5.1 Flow Analysis

The flow analysis algorithm bears a superficial resemblance to the partial evaluation algorithm in [1], but the aim is to produce a set of atoms and not a partial evaluated program as in [1].

Let P be a normal program, and G a goal. Let U be an unfolding rule. The following algorithm returns a set of atoms.

```

begin
  A[0] := the set of atoms in G
  i := 0
  repeat
    P' := a partial evaluation
          of A[i] in P using U.
    S := A[i] U
          { p(t) | B <- Q,p(t),Q' in P'
            OR
            B <- Q,not(p(t)),Q' in P',
            where p is not a built-in or
            imported predicate}
    A[i+1] := abstract(S)
    i := i+1
  until A[i] = A[i-1] (modulo variable renaming)
end

```

The set of atoms $\mathbf{A}[i]$ on termination is taken as the output of the algorithm.

In the version of the algorithm implemented in SP, the procedure is optimised, so that only the newly computed atoms are processed on each iteration. The termination condition is that no new atoms are computed on an iteration. Further details of this kind of optimisation are found in [11].

5.2 Termination of the Flow Analysis

The termination of the flow analysis depends on two things:

- the computation rule U , which should produce a finite set of resultants for each atom;
- the operation $abstract(S)$, which should ensure that a finite set of atoms is returned by the algorithm.

Regarding U , depth bounds on unfolding can be used to ensure termination, but the determinate unfolding rule (possibly with more specific resolvents) can be used safely for “reasonable” programs. A large value for the depth bound seems a good compromise: in practice it is hardly ever needed.

If the user declares atoms to be executable, care should be taken that the number of solutions of executable atoms is finite.

Successive iterations of the algorithm produce sets $A[i], A[i + 1]$, where $A[i + 1]$ differs from $A[i]$ either by adding an atom with a new characteristic tree, or by replacing an atom in $A[i]$ by some more general atom with the same characteristic tree (see Definition 4.3). An unfolding rule with a fixed depth bound ensures that there is a finite number of possible characteristic trees. Termination follows from the fact that the number of generalisations of an atom (modulo variable renaming) is finite.

5.3 Construction of Specialised Programs

Lemma 5.1 *Let the input of the algorithm be a program P , a goal G and an unfolding rule U . Assume the algorithm terminates with atoms $\mathbf{A}[i] = \mathbf{A}$; let P' be a partial evaluation of \mathbf{A} in P using U . Then $P' \cup \{G\}$ is \mathbf{A} -closed.*

PROOF. $\mathbf{A} = abstract(S)$ where S consists of atoms occurring in the bodies of clauses in a partial evaluation of \mathbf{A} in P using U (since $\mathbf{A} = \mathbf{A}[i] = \mathbf{A}[i - 1]$ at termination).

Lemma follows from the definition of $abstract$ and the fact that the head of every clause in P' is an instance of an atom in \mathbf{A} . □

Let P be a program, G a goal and U an unfolding rule. Assume the flow analysis algorithm yields a finite set of atoms \mathbf{A} . Then the following algorithm constructs a specialised program for G .

begin

P1 := a partial evaluation of \mathbf{A} in P using U .

```

R := a set of renaming definitions for A.
R1 := { A'.theta <- Q | A' <- A in R,
        A.theta <- Q in P1 }
R2 := R1 U {Ans(x1...xn) <- G}
        where x1...xn is the tuple of
        distinct variables in G.
R3 := { H <- Q' | H <- Q in R2,
        Q' is a complete renaming
        (using R) of Q}.

```

end

The output is the program R3.

Proposition 5.2 *$P \cup \{G\}$ has an SLDNF refutation with computed answer θ (resp. has a finitely failed SLDNF tree) iff $R_3 \cup \{\leftarrow Ans(\bar{x})\}$ has an SLDNF refutation with computed answer θ (resp. has a finitely failed SLDNF tree).*

PROOF. (Outline)

- $P \cup \{G\}$ has an SLDNF refutation with computed answer θ (resp. has a finitely failed SLDNF tree) iff $P \cup R \cup \{Ans(\bar{x}) \leftarrow G\} \cup \{\leftarrow Ans(\bar{x})\}$ has an SLDNF refutation with computed answer θ (resp. has a finitely failed SLDNF tree).
- Let H be any goal. $P \cup R \cup \{Ans(\bar{x}) \leftarrow G\} \cup \{H\}$ has an SLDNF refutation with computed answer θ (resp. has a finitely failed SLDNF tree) iff $P \cup R_3 \cup \{H\}$ has an SLDNF refutation with computed answer θ (resp. has a finitely failed SLDNF tree). (Use correctness of unfolding, more specific program transformation, and renaming).
- Computation of $\{\leftarrow Ans(\bar{x})\}$ calls only clauses in R_3 , built-in predicates or imported predicates (using Lemma 5.1). Hence the clauses in P are not used and can be dropped.

□

The use of the $Ans(\bar{x})$ predicate is not always satisfactory, since the specialised program should really preserve the computations of G . The $Ans(\bar{x})$ predicate was introduced primarily to simplify the presentation and correctness proof.

The following procedure can be applied to remove the Ans clause. A set of atoms is *independent* if no two atoms in the set have a common instance [20]. Let A_G be the set of atoms in G . The following procedure can be applied only if $abstract(A_G)$ is independent.

Let $A\theta$ be an atom in G and let $A' \leftarrow A$ be the renaming definition that was used to perform renaming of $A\theta$ in the above procedure.

1. Undo all renamings that used A' ;

2. Replace clauses of the form $A'\rho \leftarrow Q$, by clauses of the form $A\rho \leftarrow Q$;
3. Omit the clause $Ans(\bar{x}) \leftarrow G'$ from \bar{R} .

(A statement and proof of the correctness of this transformation is omitted for the time being.)

Example 6 *Let P be the program;*

```
Member(x, [x|y]) <-
Member(x, [y|z]) <- Member(x, z)
```

Let G be $\leftarrow \text{Member}(u, [1|w])$. A possible set of atoms returned by flow analysis is

```
{Member(u, [1|w]), Member(u, [v|w])}
```

(Note the second atom is a more specific version of $\text{Member}(x, y)$ derived during flow analysis). The specialised program produced by the procedure is:

```
Ans(u, w) <- M1(u, w)

M1(1, y) <-
M1(x, [y|z]) <- M2(x, y, z)

M2(x, x, y) <-
M2(x, w, [y|z]) <- M2(x, y, z)
```

After removing the Ans clause, the program becomes:

```
Member(1, [1|y]) <-
Member(x, [1, y|z]) <- M2(x, y, z)

M2(x, x, y) <-
M2(x, w, [y|z]) <- M2(x, y, z)
```

5.4 Cleaning Up

After the construction of the specialised program, some further transformations may be carried out in order to improve the look and efficiency of the result. These operations include:

- Unfolding calls to procedures defined by only one clause.

- Factoring out common structure. Although the renaming procedure tends to flatten out structure in the specialised program, it may sometimes happen that a variable in a clause body becomes instantiated, causing extra run time structure to be used. For example, given the clause $P(F(x)) \leftarrow Q(F(x), R(F(x)))$, the two occurrences of $F(x)$ in the body will be created as separate structures in most Prolog implementations. Transformation of the clause into $P(F(x)) \leftarrow y = F(x), Q(y), R(y)$ could improve performance, in spite of the extra unification, since y is stored only once. In SP, common structures at the argument level are factored out in this way, if desired.
- Duplicate goals in the body of a clause may be removed.

6 How to Use SP

6.1 Starting Up

After starting Prolog, the specialisation programs are read in.

```
| ?- [sp].
...

yes
| ?-
```

The specialisation tools are started by the query:

```
| ?- sp.
```

The following menu appears:

1. Read program to be specialised.
2. Specialisation.
3. Select unfolding strategy.

Choice (0 to exit):

6.2 Reading Program to be Specialised

Read in the program to be specialised by selecting 1 and then, when prompted, the name of the file containing the program. The default file name extension `.pl` can be omitted. For example:

```

Choice (0 to exit): 1

File Name: '../Exs/trans'.

.....
Finished reading ../Exs/trans.pl

```

The main menu then appears again.

6.3 Specialising With Respect to a Goal

The program read in may be specialised with respect to a query by selecting 2 from the main menu, and then, when prompted, an atomic goal. The flow analysis, program construction and cleaning up phases are indicated, and then finally, a prompt appears for a file to which the specialised program should be written. The default (carriage return) causes the result to be written to the screen. Otherwise, give the name of a file in which the result is to be written. Example:

```

Choice (0 to exit): 2

Atomic goal: transpose([_,_],_).

flow analysis...*
constructing specialised program...

transpose([A1,A2],A3)

cleaning up...
Enter filename: <CR>

Specialised Program:

transpose([],[],[]) :-
    true.
transpose([[X0|X1],[X2|X3]], [[X0,X2]|X4]) :-
    transpose([X1,X3],X4).

```

6.4 Selection of the Unfold Rule

The default unfold rule is the determinate unfolding rule with more specific clause transformations, as defined in Definition 3.6. Other unfold rules may be substituted instead. Select 3 from the main menu. At present three choices are available.

Choice (0 to exit): 3

1. Determinate with more specific transformation.
2. Determinate unfolding.
3. No unfolding.

Choice (0 for previous level):

The simpler determinate unfolding rule (Definition 3.3) can be used by selecting 2. By selecting 3, the empty unfolding rule (Definition 3.8) is read in. (Note that some program definitions are overwritten when consulting the new definitions - simply answer `p` to all Sicstus prompts). When empty unfolding is selected, the transformations are those described in [10].

6.5 Imported and Executable Atoms

As discussed in Section 2, certain predicates are regarded as imported and not available for selection during derivations. If a predicate `p(x1,...,xn)` is defined in another module, it can be declared `imported`. Add the clause

```
imported(p(x1,...,xn)).
```

to the file containing the program to be specialised.

An atom that is executable (see Section 3.4) can be declared by adding a clause

```
evaluable(p(x1,...,xn)) <- conditions(x1,...,xn).
```

to the file containing the program to be specialised, where `conditions(x1,...,xn)` define the circumstances under which atoms with predicate `p` may be executed.

6.6 Direct Calls to the Specialiser

The menu interface described above is for convenience. Sometimes it is useful to call the specialisation procedure directly from the Prolog user level. First the menu should be started in order to read in the program to be specialised, as above. Then quit the menu system. The call

```
?- sp(Goal).
```

causes the program to be specialised for the atomic goal `Goal`. The usefulness of this is that `sp(Goal)` can be called as part of a longer goal. For example:

```
?- foo(X),sp(p(X,Y)).
```

Examples of such usage can be found in the Appendix A.

7 Discussion

The SP system can be compared with program specialisation systems both for logic programs and for other languages. Since Komorowski introduced partial evaluation in logic programming [15], several systems for partial evaluation of Prolog have been reported. Lam [18] compares the results of some of these. Some aspects of SP are contained in other systems. The main points of comparison are the handling of termination, the unfolding strategy, renaming and various minor transformations.

7.1 The Role of Flow Analysis

SP is basically a two-phase specialiser, as is the *Mix* program [14]. The first phase performs analysis which is then fed into the partial evaluation phase. But in *Mix* and related systems, analysis is primarily intended to provide control information, whereas in SP it is to handle termination by obtaining a set of atoms whose partial evaluation contains only instances of those atoms.

In SP specialisation is seen as an application of a global analysis, similar to applications of abstract interpretation. The flow analysis of SP is intended to provide a set of atoms that “represent” the set of all calls that occur in the computation. This set is in general infinite, so a safe approximation and a fixpoint computation are used to get a safe approximation. The second phase is then a relatively trivial process that creates a renaming and a procedure for each distinct call. It is not surprising that the closedness property given in [20] expresses a similar requirement as the requirement of safety of an abstract interpretation. The presentation in this paper suggests that more precise representations of the set of calls (using abstract substitutions) could in principle be used instead of a set of atoms.

7.2 Control and Determinacy

In SP termination is a global problem requiring analysis, while control is a local problem whose solution depends mainly on determinacy. In some partial evaluation systems, the aim of the unfolding strategy is to unfold as much as possible while still ensuring termination. This strategy is based on the understandable assumption that any unfolding step done at compile time is one less to be done at run time. In one sense this is true, since unfolding never lengthens the successful branches of a computation tree. However the computation rule to be used at run time has to be taken into account; any unfolding performed during partial evaluation essentially “compiles in” a control choice. Thus the choices made during partial evaluation are as crucial to performance as those made at run time. It has been shown [32] that unfolding may increase backtracking and create extra choice points. Introducing disjunctions to handle choices is also a bad idea when applied blindly, since typical Prolog implementations lose indexing information in the process.

The unfolding strategy of SP is based on determinacy. Unfolding determinate sections of the computation is never wrong, no matter what run time control is used. In fact, in SP it is assumed that the Prolog left-to-right computation rule will be applied at run time, so choices at the left of a goal may also safely be unfolded. But here the issue of program size is taken into account; unfolding choice points may increase the size of the program dramatically. While this is not necessarily bad, SP makes the conservative choice not to unfold choices unless the user specifically designates an atom as executable.

Although the importance of determinacy in logic programs seems clear enough, it can be argued that it is equally important in functional and other languages, with more fundamental importance than the static-dynamic distinction widely used for control. The usefulness of unfolding static parts of the computation is precisely because in functional programs, static usually corresponds to determinate.

7.3 Renaming

The specialisation in SP includes renaming as an essential part. This is because the removal of redundant structures from the specialised program often gives significant optimisation, especially of space usage. Secondly, a theoretical advantage is that the concept of *independence* is not needed (as defined in [20] and incorporated in the algorithm in [1]).

Other partial evaluation systems have included similar renaming methods (e.g. [6], [24], [1]); the use of renaming definitions in SP clarifies and standardises the various notions of renaming.

7.4 Results

SP has been applied to a number of examples discussed in the literature. A selection of results appears in Appendix A. It appears to perform at least as well on most programs as other logic program specialisation systems that operate on declarative programs. One exception is the `contains` example discussed by [18], [17] and [26]. SP's strategy here is too conservative and does not unfold enough to give the required specialisation. The strategy of unfolding only determinate parts of the computation performs well on a wide range of examples but apparently choice points must be expanded in some cases.

Furthermore, SP's results are obtained without departing from the SLDNF framework. The well-known example of the Knuth-Morris-Pratt string matching transformation is handled, for instance, although other researchers have used constraints to process this example [30], [6].

SP makes no attempt to handle cuts, side effects and other non-declarative aspects of Prolog, as is done by Mixtus [25].

SP is not yet self-applicable, since the SP system is not written declaratively. This remains an important goal; an effort will shortly be made to rewrite SP in a declarative subset of Prolog. This will entail handling a ground representation of the object program.

Acknowledgements

The SP system has been developed over a period, and has benefited from discussions between the author and several people, including Kerima Benkerimi, Maurice Bruynooghe, Danny De Schreye and John Lloyd. The paper by Lloyd and Shepherdson [20] and subsequently the algorithm developed by Benkerimi and Lloyd [1] greatly clarified the concepts of partial evaluation. Experiments performed using SP by Carl Seghers and André de Waal have provided very useful feedback.

References

- [1] Benkerimi, K. and Lloyd, J.W.; A Procedure for Partial Evaluation of Logic Programs; in *Proceedings of the North American Conference on Logic Programming*, (eds. S. Debray and M. Hermenegildo); Austin, Texas; November 1990; MIT Press, 1990.
- [2] Bruynooghe, M., De Schreye, D., and Krekels, B.; Compiling Control; *Journal of Logic Programming* 6 (1989), pages 135-162.
- [3] de Waal, D.A. and Gallagher, J.; Specialisation of a Unification Algorithm; Presented at LOPSTR'91, Manchester University, 1991; Univ. of Bristol, Department of Computer Science, TR-91-14, 1991.
- [4] Ershov, A.P.; On the Essence of Compilation; in *Formal Descriptions of Programming Languages*; ed. E.J. Neuhold; pages 390-421; North-Holland, 1978.
- [5] Futamura, Y.; Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler; *Systems, Computers, Controls*; 2(5), pages 45-50, 1971.
- [6] Fujita, H.; An Algorithm for Partial Evaluation with Constraints; ICOT Technical Memorandum, TM-0484, August 1987.
- [7] Gallagher, J.; An Approach to the Control of Logic Programs; Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, Ireland; January, 1983.
- [8] Gallagher, J.; Transforming Logic Programs by Specialising Interpreters; in ECAI-86, Proc. of the 7th European Conference on Artificial Intelligence; Brighton, England; pages 109-122, 1986.
- [9] Gallagher, J., Codish, M., Shapiro, E.Y.; Specialisation of Prolog and FCP Programs Using Abstract Interpretation; *New Generation Computing* 6 (1988) pages 159-186.
- [10] Gallagher, J. and Bruynooghe, M.; Some Low-Level Source Transformations for Logic Programs; in *Proceedings of Meta90 Workshop on Meta Programming in Logic*, Leuven, Belgium, April 1990.
- [11] Gallagher, J. and Bruynooghe, M.; The Derivation of an Algorithm for Program Specialisation; to appear in *New Generation Computing*; earlier version appears in *7th Int. Conf. on Logic Programming*, Jerusalem, (eds. D.H.D. Warren and P. Szeredi), MIT Press, 1990.
- [12] Gallagher, J. and Hill, P.M.; An Interpreter for the Ground Representation; Technical Report, University of Bristol; (in preparation) 1991.
- [13] Jones, N.D.; Challenging Problems for Program Specialisation and Mixed Computation; in *Partial Evaluation and Mixed Computation*; (eds. D. Bjørner, A.P. Ershov and N.D. Jones); pages 225-82; North-Holland, 1988.
- [14] Jones, N.D., Sestoft, P. and Søndergaard, H.; An Experiment in Partial Evaluation: The Generation of a Compiler Generator; in *Rewriting Techniques and Applications*, (ed. N. Jouannaud), LNCS Vol. 202; pages 124-140; Springer-Verlag, 1985.

- [15] Komorowski, H.J.; Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog; in *9th ACM Symposium on Principles of Programming Languages*; Albuquerque, 1982, New Mexico, pages 255-267.
- [16] Komorowski, H.J.; Synthesis of Programs in the Framework of Partial Deduction; Report No. 81, 1989, Department of Computer Science, Åbo Akademi, Lemminkäinenkatu 14, SF-20520 Åbo, Finland.
- [17] Kursawe, P.; Pure Partial Evaluation and Instantiation; in *Partial Evaluation and Mixed Computation*; (eds. D. Bjørner, A.P. Ershov and N.D. Jones); pages 279-90; North-Holland, 1988.
- [18] Lam, J.; Control Structures in Partial Evaluation of Pure Prolog; Master's Thesis, Department of Computational Science, University of Saskatchewan, 1989.
- [19] Lloyd, J.W.; *Foundations of Logic Programming, 2nd Edition*; Springer-Verlag, 1987.
- [20] Lloyd, J.W. and Shepherdson, J.C.; Partial Evaluation in Logic Programming; University of Bristol, Department of Computer Science, TR-87-09 (1987) (revised 1989); (to appear in *Journal of Logic Programming*).
- [21] Lombardi, L.A.; Incremental Computation; in *Advances in Computers*, (eds. F.L. Alt and M. Rubinoff); Vol. 8, pages 247-333, Academic Press, 1967.
- [22] Marriott, K., Naish, L. and Lassez, J-L.; Most Specific Logic Programs; in *Proceedings of the Fifth International Conference and Symposium on Logic Programming*; Washington, Seattle; August 1988.
- [23] Mellish, C.S.; Using Specialisation to Reconstruct Two Mode Inference Systems; Dept. of Artificial Intelligence, Univ. of Edinburgh, 1990.
- [24] Safra, S., and Shapiro, E.Y.; Meta Interpreters for Real; in *Proc. IFIP'86*, Dublin, (ed. H-J. Kugler), North-Holland 1986.
- [25] Sahlin, D.; The Mixtus Approach to Automatic Partial Evaluation for Full Prolog; in *Proceedings of the North American Conference on Logic Programming*, (eds. S. Debray and M. Hermenegildo); Austin, Texas; November 1990; MIT Press, 1990.
- [26] Sahlin, D.; An Automatic Partial Evaluator for Full Prolog; Ph.D. thesis, Royal Institute of Technology, Department of Telecommunications and Computer Systems, Stockholm, Sweden; March 1991.
- [27] Seghers, C.; Compiling with Partial Evaluation; Thesis Dissertation, K.U. Leuven, Belgium, 1991.
- [28] Seki, H.; Unfold/Fold transformations of Stratified Programs; in *Proc. of the 6th Int. Conf. on Logic Programming*, (eds. G. Levi and M. Martelli), Lisbon, 1989, MIT Press, 1989.

- [29] Sestoft, P. and Zamulin, A.V.; A Bibliography on Partial Evaluation and Mixed Computation; in *Proceedings of Workshop on Partial Evaluation and Mixed Computation*, Denmark Oct. 87.
- [30] Smith, D.A.; Partial Evaluation of Pattern Matching in Constraint Logic Programming Languages; in *PEPM'91, Proc. of the ACM Symposium on Partial Evaluation and Semantics-Based Program Transformation*; SIGPLAN Notices Vol. 26, No. 9, September 1991.
- [31] Tamaki, H. and Sato, T.; OLD Resolution with Tabulation; in *Proceedings of the 3rd International Conference in Logic Programming*, LNCS 225; Springer-Verlag, 1986.
- [32] Venken, R. and Demoen, B.; A Partial Evaluation System for Prolog; Some Practical Considerations; in *Partial Evaluation and Mixed Computation*; (eds. D. Bjørner, A.P. Ershov and N.D. Jones); pages 279-90; North-Holland, 1988.

A Examples of Specialisation

The following examples are taken from the literature and the results shown are exactly as produced by SP. For each example the source program is shown, followed by the results of specialising one or more queries on the program.

A.1 Transposing a Matrix

The example first appeared in [7] and was also discussed in [8] and other papers.

```
/* Transpose a matrix */

transpose(Xs, []) :-
    nullrows(Xs).
transpose(Xs, [Y|Ys]) :-
    makerow(Xs, Y, Zs),
    transpose(Zs, Ys).

makerow([[X|Xs]], [X], [Xs]).
makerow([[X|Xs]|Ys], [X|Xs1], [Xs|Zs]) :-
    makerow(Ys, Xs1, Zs).

nullrows([]).
nullrows([[_|Ns]]) :-
    nullrows(Ns).
```

Atomic goal: transpose([_,_],_).

Specialised Program:

```
transpose([], [], []) :-
    true.
transpose([[X0|X1], [X2|X3]], [[X0, X2] | X4]) :-
    transpose([X1, X3], X4).
```

Atomic goal: transpose([[_,_] | _],_).

Specialised Program:

```
transpose([[X0, X1] | X2], [[X0 | X3], [X1 | X4]]) :-
    makerow_1(X0, [X1], X2, X3, X5),
    makerow_1(X1, [], X5, X4, X6),
    nullrows_1(X6).
makerow_1(X0, X1, [], [], []) :-
    true.
```

```

makerow_1(X0,X1,[[X2|X3]|X4],[X2|X5],[X3|X6]) :-
    makerow_1(X2,X3,X4,X5,X6).
nullrows_1([]) :-
    true.
nullrows_1([[_|X0]) :-
    nullrows_1(X0).

```

Atomic goal: transpose([[_,_],_],_).

Specialised Program:

```

transpose([[X0,X1],[X2,X3]],[[X0,X2],[X1,X3]]) :-
    true.

```

A.2 Flatten a List

Example taken from [1].

```

flatten([], []).
flatten([X|Y],Z) :-
    flatten(X,Z1),
    flatten(Y,Z2),
    append(Z1,Z2,Z).
flatten(X,[X]) :-
    number(X).

```

```

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).

```

Atomic goal: flatten([[1,2],[[3,4|_]]],_).

Specialised Program:

```

flatten([[1,2],[[3,4|X0]|X1],[1,2,3,4|X2]) :-
    flatten_1(X0,X3),
    append_1(X3,[],X4),
    flatten_1(X1,X5),
    append_1(X4,X5,X2).
flatten_1([],[]) :-
    true.
flatten_1([X0|X1],X2) :-

```

```

        flatten_1(X0,X3),
        flatten_1(X1,X4),
        append_1(X3,X4,X2).
flatten_1(X0,[X0]) :-
    number(X0).
append_1([],X0,X0) :-
    true.
append_1([X0|X1],X2,[X0|X3]) :-
    append_1(X1,X2,X3).

```

A.3 String Matching

Example discussed by [6], [18] and others. This shows how an efficient matcher implementing the Knuth-Morris-Pratt algorithm can be derived for a given ground pattern, from a naive string matching program. Note that the other authors cited have assumed that to produce the result below, some notion of constraints is needed. As is shown below, this is not the case; the result is obtainable using SLDNF computation only. Negative literals can safely be selected when ground; the predicate

```
A \== B
```

may be selected when ground and this is sufficient to obtain the required result. (It could alternatively be treated as the negation of $A = B$ in this example).

```

match(P,T) :-
    match1(P,T,P,T).

match1([],_,_,_).
match1([A|Ps],[A|Ts],P,T) :-
    match1(Ps,Ts,P,T).
match1([A|_Ps],[B|_Ts],P,[_T]) :-
    A \== B,
    match1(P,T,P,T).

```

```
Atomic goal: match([a,a,b],_).
```

Specialised Program:

```

match([a,a,b],[X0|X1]) :-
    match1_1(X0,X1).
match1_1(a,[X0|X1]) :-
    match1_2(X0,X1).
match1_1(X0,[X1|X2]) :-
    a\==X0,
    match1_1(X1,X2).

```

```

match1_2(a, [X0|X1]) :-
    match1_3(X0,X1).
match1_2(X0, [X1|X2]) :-
    a\==X0,
    a\==X0,
    match1_1(X1,X2).
match1_3(b,X0) :-
    true.
match1_3(X0,X1) :-
    b\==X0,
    match1_2(X0,X1).

```

A.4 An Interpreter for a Ground Representation

This example (from [12]) shows how specialisation of an interpreter for a ground representation of object programs can result in the elimination of practically all operations concerning the representation. In the program below, an object program atom $p(f(X),Y)$ would be represented in the meta program by a term such as $p(f(var(0)),var(1))$.

```

/* instantiate(X,Y):
   X in ground representation,
   Y a ground instance of X */

instantiate(X,Y) :-
    inst(X,Y, [],_).

inst(var(N),X, [], [(N/X)]) :-
    anyterm(X).
inst(var(N),X, [(N/X)|S], [(N/X)|S]).
inst(var(N),X, [(M/Y)|S], [(M/Y)|S1]) :-
    N \== M,
    inst(var(N),X,S,S1).
inst(T,T1,S,S1) :-
    T =.. [F|Xs],
    F \== var,
functor(T,F,N),
functor(T1,F,N),
    T1 =.. [F|Ys],
    inst_args(Xs,Ys,S,S1).

inst_args([], [], S,S).
inst_args([X|Xs], [Y|Ys], S,S2) :-
    inst(X,Y,S,S1),
    inst_args(Xs,Ys,S1,S2).

```

```

/* anyterm(X) */

imported(anyterm(_)). /* control for partial evaluation */
                      /* anyterm(_) is not unfolded      */

anyterm(X) :-
    atomic(X).
anyterm(T) :-
    T =.. [_|Xs],
    anyargs(Xs).

anyargs([]).
anyargs([X|Xs]) :-
    anyterm(X),
    anyargs(Xs).

statement_instance(S,P) :-
    member(C,P),
    instantiate(C,S).

member(X,[X|_]).
member(X,[_|Y]) :-
    member(X,Y).

demo(G,G1,P) :-
    instantiate(G,G1),
    demo1(G1,P).

demo1(true,_).
demo1((G,Gs),P) :-
    demo1(G,P),
    demo1(Gs,P).
demo1(G,P) :-
    statement_instance(statement((G :- B)),P),
    demo1(B,P).

/* Example program */

memberprog(
    [
        statement( (member(var(1),[var(1)|var(2)]) :- true)),

```

```

statement( (member(var(1),[var(2)|var(3)]) :-
member(var(1),var(3))))
] ).

```

Atomic goal: `instantiate(p(var(0),var(1),var(0)),_)`.

Specialised Program:

```

instantiate(p(var(0),var(1),var(0)),p(X0,X1,X0)) :-
anyterm(X0),
anyterm(X1).

```

```

| ?- memberprog(P),sp(demo(_,_),P)).

```

Specialised Program:

```

demo(X0,X1,[
statement( (member(var(1),[var(1)|var(2)]) :- true)),
statement( (member(var(1),[var(2)|var(3)]) :-
member(var(1),var(3))))
]) :-
inst_1(X0,X1,X2),
demo1_1(X1).

```

```

inst_1(var(X0),X1,[X0/X1]) :-
anyterm(X1).

```

```

inst_1(X0,X1,X2) :-
X0=..[X3|X4],
X3\==var,
functor(X0,X3,X5),
functor(X1,X3,X5),
X1=..[X3|X6],
inst_args_1(X4,X6,X2).

```

```

demo1_1(true) :-
true.

```

```

demo1_1((X0,X1)) :-
demo1_1(X0),
demo1_1(X1).

```

```

demo1_1(member(X0,[X0|X1])) :-
anyterm(X0),
anyterm(X1),
demo1_1(true).

```

```

demo1_1(member(X0,[X1|X2])) :-
    anyterm(X0),
    anyterm(X1),
    anyterm(X2),
    demo1_1(member(X0,X2)).

inst_args_1([],[],[]) :-
    true.
inst_args_1([X0|X1],[X2|X3],X4) :-
    inst_1(X0,X2,X5),
    inst_args_2(X1,X3,X5,X4).
inst_args_2([],[],X0,X0) :-
    true.
inst_args_2([X0|X1],[X2|X3],X4,X5) :-
    inst_4(X0,X2,X4,X6),
    inst_args_2(X1,X3,X6,X5).
inst_4(var(X0),X1,[],[X0/X1]) :-
    anyterm(X1).
inst_4(var(X0),X1,[X0/X1|X2],[X0/X1|X2]) :-
    true.
inst_4(var(X0),X1,[X2/X3|X4],[X2/X3|X5]) :-
    X0\==X2,
    inst_4(var(X0),X1,X4,X5).
inst_4(X0,X1,X2,X3) :-
    X0=..[X4|X5],
    X4\==var,
    functor(X0,X4,X6),
    functor(X1,X4,X6),
    X1=..[X4|X7],
    inst_args_2(X5,X7,X2,X3).

```

A.5 Compiling An Expression Interpreter

An evaluator for arithmetic expressions, with an argument simulating memory giving the bindings of variables, is specialised with respect to a given expression. This could be regarded as part of the compilation of an imperative language. (See [27]).

```

eval(X + Y, Env,Z ) :-
    eval(X,Env,Z1),
    eval(Y,Env,Z2),
    Z is Z1+Z2 .
eval(X * Y, Env,Z ) :-
    eval(X,Env,Z1),
    eval(Y,Env,Z2),
    Z is Z1*Z2 .

```

```

eval(X / Y, Env,Z ) :-
    eval(X,Env,Z1),
    eval(Y,Env,Z2),
    Z is Z1/Z2 .
eval(X - Y, Env,Z ) :-
    eval(X,Env,Z1),
    eval(Y,Env,Z2),
    Z is Z1-Z2 .
eval(- Y, Env,Z ) :-
    eval(Y,Env,Z2),
    Z is -Z2 .
eval(+ Y, Env,Z ) :-
    eval(Y,Env,Z).
eval(int(X),_,X).
eval(X,Env,Z) :-
    atom(X),
    lookup(X,Env,Z).

lookup(X,[val(X,Y)|_],Y).
lookup(X,[_|Xs],Y) :-
    lookup(X,Xs,Y).

```

Atomic Goal: eval(x+y*int(3)/z,_,_).

Specialised Program:

```

eval(x+y*int(3)/z,[X0|X1],X2) :-
    lookup_1(x,X0,X1,X3),
    lookup_1(y,X0,X1,X4),
    X5 is X4*3,
    lookup_1(z,X0,X1,X6),
    X7 is X5/X6,
    X2 is X3+X7.
lookup_1(X0,val(X0,X1),X2,X1) :-
    true.
lookup_1(X0,X1,[X2|X3],X4) :-
    lookup_1(X0,X2,X3,X4).

```

B Unfoldability Conditions for Built-ins

This appendix contains the definition of the procedure in SP that determines whether a call to a built-in predicate is evaluable.

```
/* unfoldability conditions for Sicstus builtins */

unfoldable(clause(X,_)) :-
    nonvar(X).
unfoldable(predicate_property(X,_)) :-
    nonvar(X).
unfoldable(_ is Y) :-
    arithexpr(Y).
unfoldable(_ = _).
unfoldable('C'(_,_,_)).
unfoldable(true).
unfoldable(X =.. _) :-
    nonvar(X).
unfoldable(_ =.. [F|Args]) :-
    atom(F),
    completeList(Args).
unfoldable(X < Y) :-
    arithexpr(X),
    arithexpr(Y).
unfoldable(X > Y) :-
    arithexpr(X),
    arithexpr(Y).
unfoldable(X =< Y) :-
    arithexpr(X),
    arithexpr(Y).
unfoldable(X >= Y) :-
    arithexpr(X),
    arithexpr(Y).
unfoldable(X @< Y) :-
    ground(X),
    ground(Y).
unfoldable(X @> Y) :-
    ground(X),
    ground(Y).
unfoldable(X @=< Y) :-
    ground(X),
    ground(Y).
unfoldable(X @>= Y) :-
    ground(X),
    ground(Y).
unfoldable(X == Y) :-
```

```

        nonvar(X),
        nonvar(Y),
        X =.. [F|_],
        Y =.. [G|_],
        F \== G,
        !.
unfoldable(X == Y) :-
    ground(X),
    ground(Y).
unfoldable(X \== Y) :-
    X == Y,
    !.
unfoldable(X \== Y) :-
    ground(X),
    ground(Y),
    !.
unfoldable(real(X)) :-
    nonvar(X).
unfoldable(integer(X)) :-
    nonvar(X).
unfoldable(number(X)) :-
    nonvar(X).
unfoldable(atomic(X)) :-
    nonvar(X).
unfoldable(atom(X)) :-
    nonvar(X).
unfoldable(ground(X)) :-
    ground(X).
unfoldable(not(X)) :-
    ground(X).
unfoldable(\+(X)) :-
    ground(X).
unfoldable(fail).
unfoldable(functor(X,_,_)) :-
    nonvar(X).
unfoldable(functor(_,Y,Z)) :-
    integer(Z),
    atom(Y).
unfoldable(arg(X,Y,_)) :-
    integer(X),
    nonvar(Y).
unfoldable(call(X)) :-
    unfoldable(X).

completeList(X) :-
    X == [].

```

```

completeList(Xs) :-
    nonvar(Xs),
    Xs = [_|Xs1],
    completeList(Xs1).

ground(T) :-
    nonvar(T),
    atomic(T),
    !.
ground(T) :-
    nonvar(T),
    T =.. [_|Args],
    groundlist(Args).

groundlist([]).
groundlist([T|Ts]) :-
    ground(T),
    groundlist(Ts).

arithexpr(T) :-
    nonvar(T),
    number(T),
    !.
arithexpr(T) :-
    nonvar(T),
    T =.. [Op|Args],
    memb1(Op, ['+', '-', '*', '/', '//', 'mod', '/', '\',
              'integer', 'float',
              '^', '>>', '<<']),
    arithexprlist(Args).

arithexprlist([]).
arithexprlist([T|Ts]) :-
    arithexpr(T),
    arithexprlist(Ts).

memb1(X, [X|_]) :-
    !.
memb1(X, [_|Xs]) :-
    memb1(X, Xs).

```