Using Regular Approximations for Generalisation During Partial Evaluation

John P. Gallagher
University of Bristol
Department of Computer Science
Bristol BS8 1UB, UK
john@cs.bris.ac.uk

Julio C. Peralta
University of Bristol
Department of Computer Science
Bristol BS8 1UB, UK
jperalta@cs.bris.ac.uk

ABSTRACT

On-line partial evaluation algorithms include a generalisation step, which is needed to ensure termination. In partial evaluation of logic and functional programs, the usual generalisation operation applied to computation states is the most specific generalisation (*msg*) of expressions. This can cause loss of information, which is especially serious in programs whose computations first build some internal data structure, which is then used to control a subsequent phase of execution - a common pattern of computation. If the size of the intermediate data is unbounded at partial evaluation time then the *msg* will lose almost all information about its structure. Hence the second phase of computation cannot be effectively specialised.

In this paper a generalisation based on regular approximations is presented. Regular approximations are recursive descriptions of term structure closely related to tree automata. A regular approximation of computation states can be built during partial evaluation. The critical point is that when generalisation is performed, the upper bound on regular descriptions can be combined with the *msg*, thus preserving structural information including recursively defined structure. The domain of regular approximations is infinite and hence a widening is incorporated in the generalisation to ensure termination. An algorithm for partial evaluation of logic programs, enhanced with regular approximations, along with some examples of its use will be presented.

1. INTRODUCTION

Partial evaluation algorithms involve a generalisation step, which is a source of imprecision but is essential if termination is to be ensured. Loss of precision arises because the generalisation step can discard aspects of computation states that would have allowed specialisation in subsequent phases of partial evaluation. As will be seen, the commonly used *most specific generalisation* (or *msg*) [25] is rather prone to throw away potential specialisations. A generalisation based on regular approximations, which can preserve much more structural information, is presented here.

This paper is mainly concerned with on-line partial evaluation, where generalisation is applied on the fly as computation states are encountered. In off-line partial evaluation generalisation arises from an abstraction applied to all computation states. Regular approximations could be applied as the abstraction in off-line partial evaluation but is not discussed in this paper. Related ideas for binding-time analysis using tree grammars have been used by Mogensen [21]. Also, regular tree grammars were used by Jones and Muchnick [11] to analyse interprocedural data flow for programs with recursive data structures such as the structure of the stack of activation records in a language interpreter.

In on-line partial evaluation generalisation is applied whenever some state is recognised as being a version, according to some criterion of similarity, of a previously produced state (possibly an ancestor in the computation). Partial evaluation is then continued using the generalisation of the two states. Similarity criteria have been based on homeomorphic embedding, neighbourhoods, trace terms, loop prevention strategies and related techniques [7; 14; 16; 26; 28; 29].

1.1 Connections Between Abstract Interpretation and Partial Evaluation

Several authors have made the connection between generalisation in partial evaluation and frameworks for abstract interpretation [3; 5; 10; 15; 24]. The notion of similarity of computation states can be linked to that of an abstract domain for describing computation states, while generalisation corresponds to the computation of an upper bound of abstract states. Although this correspondence is clear, it has proved difficult so far to identify general-purpose abstract domains suitable for partial evaluation. In other words, a good abstraction seems to be dynamic and dependent on a particular computation.

The approach based on abstract interpretation promises in the long run to unite on-line and off-line partial evaluation, and provide a coherent framework for treating control of unfolding and polyvariance. However, it is not yet obvious how to contruct an abstract domain incorporating dynamic features such as homeomorphic embeddings, loop detection methods, trace terms and so on: yet these concepts seem to be the right ones for getting good specialisation.

In this paper the key concepts of abstraction and upper bound are introduced into an on-line partial evaluation algorithm. This has been suggested before [3; 5] but in this

paper a specific abstraction based on regular sets of terms is used and tested. Regular approximations are especially useful because they relate closely with the concrete representations in an on-line partial evaluation algorithm.

1.2 Regular Approximation of Intermediate States

One of the main motivations for improving the generalisation is provided by a common pattern of program execution; a first phase of computation builds some internal data structure, which is then used to drive a subsequent phase of execution. Typical examples include the following.

- A parser-generator constructs a parse tree and symbol table, which are then used to guide code generation.
- An interpreter for a language with procedure calls typically builds up code continuations as a stack of pieces of code that are executed on procedure exit.
- Graph-processing algorithms often require a preprocessing step to render a graph in a convenient form first.

When attempting to specialise such programs, the successful specialisation of the second phase depends completely on propagating the results of specialising the first phase. If the number of possible results of the first stage (at partial evaluation time) is unknown, then generalisation is applied to represent the set of results finitely. This generalisation might well have thrown out the information needed to specialise the second phase.

For instance, the interpreter for procedures mentioned above will have a continuation stack of unknown size where recursive procedures are allowed, if the depth of recursion is unknown at partial evaluation time. The partial evaluator must generalise an infinite number of possible code continuations. Using a generalisation based on *msg* the information about continuations below some finite depth will be lost. To illustrate this, suppose the continuation stack is represented as a nested term; the *msg* of `cont(code1, code2)` and `cont(code1, cont(code1, code2))` is `cont(code1, X)`. Here `X` is a variable and destroys any chance of specialising the interpreter, since after some procedure exit the interpreter continues with unknown code.

To anticipate the solution adopted in this paper, the continuation stacks in this example will be generalised using a representation that preserves the recursive structure. The continuation will be represented as `X` such that `t(X)` holds, where `t` is defined by the following regular logic program.

```
t(code2) <-
t(cont(Y,Z) <- t1(Y), t(Z)
t1(code1) <-
```

The regular unary predicates can be thought of as *constraints* on the values of variables. Then the unfolding mechanism during partial evaluation will be augmented to check that the terms are consistent with these regular constraints.

2. REGULAR APPROXIMATIONS

From now on the terminology and notation of logic programming are adopted. The concepts of regular approximation and type graphs[9; 8] are independent of any particular programming language, however.

A *Regular Unary Logic (RUL)* program is a set of clauses of the following form.

$$t_0(f(X_1, \ldots, X_n)) \leftarrow t_1(X_1), \ldots, t_n(X_n) \ (n \geq 0)$$

where $X_1, \ldots, X_n$ are distinct variables. A *deterministic* or *canonical* RUL program is an RUL program in which no two clause heads have a common instance.

It can be shown that RUL programs correspond to top-down deterministic tree automata [1]; that is they can be used as recognisers for term languages. Given any ground atom for a unary predicate t, it is decidable whether it is accepted (that is, succeeds) in a given RUL program. Conversely, every unary predicate t in an RUL program P can be identified with the set of ground terms $f(s_1, \ldots, s_n)$ for which $t(f(s_1, \ldots, s_n))$ succeeds in P. This will be called the *success set* of t in P. It is decidable whether the success set of t in P is empty. A set of terms is regular (top-down deterministic regular) if and only if it can be defined as the success set of a unary predicate in an RUL (deterministic RUL) program. The special unary predicate `any` has the set of all terms as its success set.

2.1 Intersection and Tuple-Distributive Union of Regular Sets

The intersection and union of regular sets of terms have direct counterparts in RUL programs. If t_1 and t_2 are unary predicates in an RUL program P with success sets S_1 and S_2 respectively, then there is an RUL program P' and predicates t_3, t_4 such that the success set of t_3 in P' is $S_1 \cap S_2$ and the success set of t_4 in P' is $S_1 \cup S_2$.

However, the union of predicates in deterministic RUL programs does not in general result in deterministic RUL programs. For instance, the sets $\{f(a,b)\}$, $\{f(c,d)\}$ and their union $\{f(a,b), f(c,d)\}$ could be represented by t_1, t_2 and t_3 respectively below.

```
t1(f(X,Y)) <- s1(X), s2(Y)
s1(a) <-
s2(b) <-

t2(f(X,Y)) <- r1(X), r2(Y)
r1(c) <-
r2(d) <-


t3(f(X,Y)) <- s1(X), s2(Y)
t3(f(X,Y)) <- r1(X), r2(Y)
```

The clauses for t_3 are non-deterministic since the heads for clauses defining `t3` have a common instance. The tuple-distributive union of two deterministic predicates is given by predicate t_4 below.

```
t4(f(X,Y)) <- w1(X), w2(Y)
w1(a) <-
w1(c) <-
```

```
w2(b) <-
w2(d) <-
```

The tuple-distributive union is not defined formally here [20; 30]. Notice that the success set of t_4 includes $f(a,d)$ and $f(c,b)$ as well as the union, but it is the smallest set containing the union that can be represented using a deterministic RUL program. Thus unlike string languages and conventional finite automata, in which a non-deterministic automaton can be translated to an equivalent deterministic one, determinacy is a restriction in tree automata. On the other hand the intersection of deterministic predicates is itself deterministic.

Given any term $f(\bar{t})$ (where $\bar{t}$ is a tuple of arguments) there exists an RUL program with some unary predicate w such that the success set of w in the program is the set of all instances of $f(\bar{t})$. This is called the *RUL representation of an atom*. For example, the RUL representation of the term $f(g(a), h(X))$ is as follows.

```
t(f(X,Y)) <- w1(X), w2(Y)
w1(g(X)) <- w3(X)
w2(h(X)) <- any(X)
w3(a) <-
```

2.2 Widening

The set of all deterministic regular sets of terms over a finite signature forms a lattice with the subset ordering. The join operation is the tuple-distributive union. Clearly the lattice has infinite height. In static analyses over the lattice monotonically increasing sequences representing successive approximations are generated. Thus a widening is needed in order to ensure convergence of such sequences. Several widenings have been suggested [9; 2; 6; 8], and their properties analysed [19]. Informally, a widening is employed to introduce recursive definitions into RUL programs. For instance the sequence of predicates representing the sets $\{[]\}, \{[], [a]\}, \{[], [a], [a,a]\}, \ldots$ could be widened to the set defined by predicate $t1$ below.

```
t1([]) <-
t1([X|Y]) <- s(X), t1(Y)
s(a) <-
```

The idea of widening is that the "growing" list is detected by comparing successive elements of the sequence; a recursive definition is introduced accordingly. In our implementation, recursive approximations are introduced by searching for a predicate t_1 that "depends on" another predicate t_2 such that the success set of t_2 is contained in that of t_1, and t_1 has the same function symbols in its clause head as $t2$ does. An illustration is provided by the representation of the set $\{[], [a], [a,a]\}$.

```
t1([]) <-
t1([X|Y]) <- s(X), t2(Y)

t2([]) <-
t2([X|Y]) <- s(X), t3(Y)

t3([]) <-

s(a) <-
```

Here $t1$ depends on (calls) $t2$ and the success set of $t2$ is $\{[], [a]\}$ which is contained in the success set of $t1$. Hence, we would form a recursive definition by replacing the occurrence of $t2$ by $t1$ in the body of the clause for $t1$, giving the recursive definition of $t1$ shown above.

This method was used [27] but it is in fact not a true widening as it does not guarantee convergence, as proved by Mildner [19]. However, as noted by Mildner, it appears to converge in all practical cases. A true widening for type graphs such as the one invented by Van Hentenryck *et al.* [8] could be used to guarantee termination in all cases.

3. THE BASIC PARTIAL EVALUATION ALGORITHM

A basic partial evaluation algorithm [5] takes as input a program and a goal and computes a set of atoms. Each iteration of the main loop of the program applies unfolding (partial evaluation) to the atoms in the current set, yielding a set of *resultants*. The atoms in the resultants are generalised (to ensure termination) and any new atoms (after generalisation) are added to the current set.

The output set of atoms at termination is closed [18], in the sense that applying one iteration of the algorithm to the set yields only atoms in the set. The algorithm is parameterised by an unfolding rule (the local control) and a generalisation operation (the global control). The latter operation determines the polyvariance (number of different versions of predicates) in the partially evaluated program, as well as ensure termination, since it incorporates some similarity criterion which is used for deciding how many versions of each program point appear in the specialised program.

The algorithm is naive in the sense that the i^{th} iteration applies partial evaluation to the whole set of atoms A_i. In practice only the new atoms (those generated for the first time on the previous iteration) need to be unfolded, but this obscures the presentation of the algorithm.

The algorithm presented below is an enhancement of the basic algorithm, in which the set of atoms is replaced by a set of objects that will be called *R-atoms*.

3.1 R-Atoms and their Representation

An *R-atom* is, in effect, an atom with some regular constraints on its arguments. An R-atom is represented by a set of clauses $\{p(\bar{t}) \leftarrow w_1(X_1), \ldots, w_k(X_k)\} \cup R$ where $\bar{t}$ is a tuple of terms, $X_1, \ldots, X_k$ are the distinct variables in $\bar{t}$ and R is a deterministic RUL program defining the predicates $w_1, \ldots, w_k$ (and any of their subsidiary predicates). Note that the clause $p(\bar{t}) \leftarrow w_1(X_1), \ldots, w_k(X_k)$ above is not itself a regular unary clause, since p is not in general a unary predicate but rather a predicate from the program being partially evaluated. Furthermore, the arguments of p (namely $\bar{t}$) can be terms of any depth, and variables can appear in $\bar{t}$ more than once.

An R-atom of this form represents any atom $p(\bar{t})\theta$ where the substitution θ is $\{X_1/s_1, \ldots, X_k/s_k\}$ and $w_j(s_j)$ is in the success set of w_j in R, for $1 \leq j \leq k$. The atom $p(\bar{t})$ will

INPUT: a program P and atomic goal G
OUTPUT: a set of R-atoms

```
begin
    A := the R-atom corresponding to G
    S_0 := {A}
    i := 0
    repeat
        S^new := unfold(S_i)
        S_{i+1} := generalise(S^new, S_i)
        i := i + 1
    until S_i = S_{i-1}
end
```

Figure 1: Basic PE Algorithm Modified for R-atoms

be called the *skeleton* of the R-atom, and the clause $p(\bar{t}) \leftarrow w_1(X_1), \ldots, w_k(X_k)$ will be called the *skeleton clause*. The conjunction $w_1(X_1), \ldots, w_k(X_k)$ can be considered as "regular constraints" on the head atom $p(\bar{t})$.
Any atom can be represented as an R-atom. Let $p(\bar{t})$ be an atom containing variables $Y_1, \ldots, Y_k$. The R-atom for $p(\bar{t})$ is given by the single (skeleton) clause

$$p(\bar{t}) \leftarrow any(Y_1), \ldots, any(Y_k)$$

For convenience each R-atom is self-contained in this presentation. This is, all the RUL clauses associated with the skeleton clause of an R-atom are contained within the R-atom. In practice different R-atoms could share RUL clauses. However, a global RUL store for all R-atoms introduces complications when widening, since widening can apply to some predicate arguments (the ones that are "growing") but not to others.

3.2 Partial Evaluation Extended with Regular Approximation

The changes to the basic algorithm all follow from the modification of atoms to R-atoms. In brief, the changes are as follows: each of these enhancements is described fully and illustrated later.

- Firstly the unfolding operation applied to atoms in the basic algorithm is altered to apply to R-atoms. Unfolding is applied to the head of the skeleton clause of the R-atom. Any substitutions that arise during unfolding are applied to the body of the skeleton clause, and the consistency of these substitutions with the corresponding RUL program is checked.
- Secondly, each iteration of the algorithm generates a new set of R-atoms by projecting the regular constraints onto the atoms resulting from unfolding.
- Thirdly, in the generalisation operation, R-atoms are compared according to the similarity criterion and the upper bound is computed for similar R-atoms, making use of the tuple-distributive upper bound, with widening applied to the result.

The enhanced algorithm has the form shown in Figure 1. Let S_1, S_2 be sets of R-atoms. The functions **unfold**(S_1) and **generalise**(S_1, S_2) are operations on sets of R-atoms. They are explained in Sections 3.3 and 3.5.

3.3 Unfolding R-Atoms

Let A be an R-atom with skeleton clause

$$p(\bar{t}_1) \leftarrow w_1(Y_1), \ldots, w_k(Y_k).$$

To unfold A, the atom $p(\bar{t})$ is first unfolded finitely in the usual way, yielding resultants $p(\bar{t})\theta_1 \leftarrow Q_1, \ldots, p(\bar{t})\theta_m \leftarrow Q_m$.
The j^{th} resultant $p(\bar{t})\theta_j \leftarrow Q_j$ is inconsistent if the constraint $(w_1(Y_1), \ldots, w_k(Y_k))\theta_j$ has no solution in the RUL program contained in the R-atom. This can be tested by unfolding $(w_1(Y_1), \ldots, w_k(Y_k))\theta_j$ with the RUL clauses until no constant or function symbols appear in the conjunction and then checking the non-emptiness of the intersection of any sets of unary atoms containing the same variable. If some such intersection is empty then the unfolding fails. Thus the regular approximation in the R-atom prunes out those resultants that are inconsistent with the regular descriptions.

Example

Let an R-atom be given as follows; the first clause is the skeleton clause.

```
p(X,Y) <- t1(X), t2(Y)

t1(a) <-
t1(f(X)) <-  t1(X)

t2(b) <-
t2(f(X)) <- t2(X)
```

Suppose `p(X,Y)` is unfolded yielding the resultants

```
p(Z,Z) <- q(Z)
p(f(X1),f(Y1)) <- p(X1,Y1)
```

The corresponding substitutions `X/Z, Y/Z` and `X/f(X1), Y/f(Y1)` are applied to the regular constraint (that is, the body of the skeleton clause) `t1(X), t2(Y)`, giving `t1(Z), t2(Z)` and `t1(f(X1)), t2(f(Y1))`. The first of these is inconsistent as the intersection of `t1` and `t2` is empty. (Note that a predicate `t` defined by the single clause form `t(f(X)) <- t(X)` is empty since no atom for `t` succeeds). The second resultant is consistent since `t1(f(X1)), t2(f(Y1))` unfolds to `t1(X1), t2(Y1)` and this has a solution (such as `X1/a, Y1/b`. Hence only the second resultant would be returned.
Let A be an R-atom. Define **resultants**(A) to be the set of resultants obtained by unfolding A and eliminating inconsistent resultants. The function is extended to apply to sets of R-atoms, where **resultants**$(S) = \bigcup\{$**resultants**$(A) \mid A \in S\}$.

3.4 Projecting R-Atoms from Resultants

A set of R-atoms is derived from the resultants. Let B_j be an atom in the body of a resultant $p(\bar{t})\theta_1 \leftarrow B_1, \ldots, B_m$. Then an R-atom for B_j is derived by projecting the regular constraints associated with the resultant onto the variables of B_j.

Example

Let `<- q(a,X),r(Y,b)` be the body of a resultant and suppose the unfolded regular constraint for this resultant is `t1(X),t2(X),t3(Y)` where the RUL program defining the unary predicates is:

```
t1(a) <-
t1(b) <-
t1(f(X)) <- t1(X)

t2(b) <-
t2(c) <-
t2(f(X)) <- t2(X)

t3(c) <-
```

Then the R-atom for `q(a,X)` is as follows. `t4` is the intersection of `t1` and `t2`.

```
q(a,X) <- t4(X)

t4(b) <-
t4(f(X)) <-  t4(X)
```

The R-atom for `r(Y,b)` is as follows.

```
r(Y,b) <- t3(Y)

t3(c) <-
```

The operator **unfold**(S) in Figure 1 where S is a set of R-atoms can now be defined. **unfold**(S) is the set of all R-atoms projected from the resultants in **resultants**(S).

3.5 Upper Bound of R-Atoms

Let A_1 and A_2 be two R-atoms with skeleton clauses $p(\bar{t}_1) \leftarrow w_1(Y_1), \ldots, w_k(Y_k)$ and $p(\bar{t}_2) \leftarrow v_1(Z_1), \ldots, v_l(Z_l)$ respectively where the skeleton clauses are renamed apart so that they have no variable in common. Assume that the upper bound is computed only between R-atoms whose skeletons have the same predicate in the head. Let $\hat{A}$ be an *msg* of $p(\bar{t}_1)$ and $p(\bar{t}_2)$, such that $\hat{A}$ contains no variable in common with $p(\bar{t}_1)$ and $p(\bar{t}_2)$. Then there are substitutions θ_1 and θ_2 such that $\hat{A}\theta_1 = p(\bar{t}_1)$ and $\hat{A}\theta_2 = p(\bar{t}_2)$. Furthermore every variable in $\hat{A}$ is bound in both θ_1 and θ_2.

For each variable X in $\hat{A}$, let X/r_1 and X/r_2 be bindings in θ_1 and θ_2 respectively. Now r_1 and r_2 are subterms of $p(\bar{t}_1)$ and $p(\bar{t}_2)$ respectively, possibly containing variables whose values are described by (some of) the unary predicates $\{w_1, \ldots, w_k, v_1, \ldots, v_l\}$. Thus r_1 and r_2 each corresponds to a regular set of terms. For each such pair of terms, an RUL program with a new unary predicate t_X can be constructed, making use of the definitions of $\{w_1, \ldots, w_k, v_1, \ldots, v_l\}$, such that the success set of t_X is the tuple-distributive union of the sets corresponding to r_1 and r_2.

The procedure for constructing the definition of t_X first constructs two clauses

$$t_{X_1}(r_1) \leftarrow w_{p_1}(Y_{p_1}), \ldots, w_{p_r}(Y_{p_r})$$

and

$$t_{X_2}(r_2) \leftarrow v_{q_1}(Z_{q_1}), \ldots, v_{q_m}(Z_{q_m})$$

where $Y_{p_1}, \ldots, Y_{p_r}$ and $Z_{q_1}, \ldots, Z_{q_m}$ are the variables occurring in r_1 and r_2 respectively. These clauses can then be converted to RUL form in a straightforward way, and the tuple-distributive union of t_{X_1} and t_{X_2} can be computed.

Example

First a trivial example is shown to give the basic idea. Let two R-atoms be given as follows.

```
p(a) <-

p(b) <-
```

Then their upper bound is given by the R-atom below.

```
p(X) <- t(X)

t(a) <-
t(b) <-
```

A somewhat more complex example illustrates some of the more subtle features of the upper bound.

Example

Let two R-atoms be given as follows.

```
% first R-atom
p(f(a),X,a) <- t1(X)

t1([]) <-
t1([X|Xs]) <- t2(X), t1(Xs)

t2(a) <-

% second R-atom
p(f(f(a)),Z,f(a)) <- t3(Z)

t3([]) <-
t3([X|Xs]) <- t4(X), t3(Xs)

t4(b) <-
```

The *msg* of `p(f(a),X,a)` and `p(f(f(a)),Z,f(a))` is `p(f(Y),W,Y)`. The bindings for `Y` to obtain the original atoms are `Y/a` and `Y/f(a)`; the bindings for `W` are `W/X` and `W/Z`. The upper bound of the terms bound to `Y` is given by predicate `t5`.

```
t5(a) <-
t5(f(X)) <- t6(X)
t6(a)
```

The upper bound of the terms bound to `W` is computed by finding the upper bounds of the predicates `t1` and `t3`. This gives the final R-atom as follows.

```
p(f(Y),W,Y) <- t5(Y),t7(W)

t5(a) <-
t5(f(X)) <- t6(X)
t6(a)

t7([]) <-
t7([X|Xs]) <- t8(X), t7(Xs)

t8(a) <-
t8(b) <-
```

The example shows that, although the tuple-distributive union discards some argument dependencies, the *msg* retains some dependency information. Thus, the upper bound includes atoms such as `p(f(a),[b,a],a)` which was not in either of the original R-atoms. However the repeated occurrence of `Y` returned by the *msg* ensures that the same term has to occur within the first and third arguments.

The operation **generalise** in the algorithm in Figure 1 is defined as follows. Let S be a set of R-atoms: partition S into a finite number of disjoint non-empty sets $\{S_1, \ldots, S_k\}$. Each partition consists of "similar" R-atoms - the partitioning determines the polyvariance of the final specialised program - and within each partition the R-atoms have the same predicate. Then $\mathbf{generalise}(S, S') = \mathbf{widen}(\{A_1, \ldots, A_k\}, S')$ where $A_1, \ldots, A_k$ are the upper bounds of $S_1, \ldots, S_k$ respectively and **widen** is a widening operator. S' is the set of R-atoms returned by the previous iteration of the algorithm.

4. EXAMPLE

The example in this section shows specialisation of an interpreter with respect to an imperative program in a small imperative language with assignments, if-then-else statements, and procedures with parameters by value. The interpreter describes a small-step operational semantics of this language as described earlier [23] using a variable environment, the program abstract syntax tree, and a stack of activation records. The language is purposefully simple to focus on what is new in the approach to partial evaluation, namely regular approximations, rather than to obscure it with complex parameter passing methods, scope rules and aliasing. A `statement(E,P,St)` predicate defines the meaning of program `P`[1] with current variable environment `E`[2] and stack of activation records `St`. Next, the logic program defining the semantics[3] of our imperative language is presented.

```
statement(Env,[],fin) <-
statement(Env,[skip|Prg],St) <-
      statement(Env,Prg,St)
statement(Env,[assign(X,Expr)|Prg],St) <-
      a_expr(Env,Expr,Vl),
      update(Env,Env1,X,Vl),
      statement(Env1,Prg,St)
statement(Env,[if(B_test,S1,S2)|Prg],St) <-
      b_expr(Env,B_test,true),
      compose(S1,Prg,SPrg),
      statement(Env,SPrg,St)
statement(Env,[if(B_test,S1,S2)|Prg],St) <-
      b_expr(Env,B_test,false),
      compose(S2,Prg,SPrg),
      statement(Env,SPrg,St)
statement(Env,[call(Prc_n,Arg)|Prg],St) <-
      address(Prg,Ad),
      code(Prc_n,proc(Actls,Prc_b)),
      a_expr(Env,Arg,Vl),
      vars(Env,Env1,Actls,Vl,R),
      statement(Env1,Prc_b,fr(R,Ad,St))
statement(Env,[rtn|_],fr(R,Ad,St)) <-
      restore(Env,Env1,R),
      code_id_cont(Env1,Ad,St)
```

Every time a procedure call is executed a frame is pushed onto the stack, containing information about the current variables and the address of the next statement where execution resumes upon return from the called procedure. In addition, the value of the formals is passed to the actuals and variable scope is updated accordingly. The following clauses define some of the predicates needed for specialisation. Henceforth, they are added to the interpreter above.

```
address([],1) <-
address([assign(f,times(f,a)),rtn],2) <-

code(fac,proc(a,
      [if(gt(n,0),[assign(n,minus(n,1)),
                   call(fac,n),
                   assign(f,times(f,a))],
                   [skip]),
       rtn])) <-

code_id_cont(N,1,TrP) <-
 statement(N,[],TrP)
code_id_cont(N,2,TrP) <-
 statement(N,[assign(f,times(f,a)),rtn],TrP)
code_id_cont(N,3,TrP) <-
 statement(N,[assign(f,plus(f,a))],TrP)

...(possibly other continuation definitions)
```

Now specialise the interpreter with respect to a procedure call (factorial) with input environment having three variables, namely `n`, `f` and `a` and initial value `1` and `12` for `f` and `a`, respectively; and `fin` as initial value of the activation record stack. Such information is encoded into the query `<-statement([_,1,12],[p(1,call(fac,n))],fin)`. After specialisation with respect to the query above, the following residual program is obtained.

```
statement(X1) <-
      statement_2(X1,12,1,fin)
statement_2(X1,X2,X3,X4) <-
      X1>0,
      statement_3(X1,X2,X3,X4)
statement_2(X1,X2,X3,X4) <-
      X1=<0,
      statement_4(X1,1,X1,fr(a(X2),X3,X4))
statement_3(X1,X2,X3,X4) <-
      X5 is X1-1,
      statement_5(X5,X1,X2,X3,X4)
statement_4(X1,X2,X3,fr(a(X4),X5,X6)) <-
      code_id_cont_8(X1,X2,X4,X5,X6)
statement_5(X1,X2,X3,X4,X5) <-
      statement_2(X1,X2,2,fr(a(X3),X4,X5))
statement_7(X1,X2,X3,X4) <-
      X5 is X2*X3,
      statement_4(X1,X5,X3,X4)
code_id_cont_8(X1,X2,X3,1,fin) <-
      write([X1,X2,X3])
code_id_cont_8(X1,X2,X3,2,X4) <-
      statement_7(X1,X2,X3,X4)
```

[1] The program is given as an abstract syntax tree.

[2] Note that the output variable environment is not observable. However, the meta-interpreter can easily be modified to show the output variable environment upon exit of program execution.

[3] For the sake of brevity we omit the definition of some of the environment handling predicates.

Existing specialisers [13; 26] are unable to obtain a satisfactory specialisation in the presence of accumulators, namely the stack of activation records. That is, information stored in the stack of activation records would be lost upon generalisation, and the residual program would be too general, potentially containing the whole interpreter. In this example the residual program does not contain code for the `code_id_cont(_,3,_)` because the procedure factorial does not contain the statements that this continuation defines. By contrast, specialisers based on plain generalisation include such a redundant code, form the point of view of the specialisation query. In this case regular approximations prevent information loss during generalisation.

5. CONCLUSIONS

The work presented here is an advance in itself, but also illustrates a more general framework for partial evaluation in which abstractions and constraints can be integrated within a "concrete" partial evaluation algorithm. The abstract domain (regular approximations in this case) are used to prune resultants and to preserve information during generalisation steps. Arithmetic constraints were already used in a similar role in other systems [12], [22]. In general an abstract domain is not defined over the same signature as the concrete program; for instance, the domain might be modes or types. In such cases, a concrete substitution arising during unfolding a concrete atom would have to be abstracted before checking its consistency with the abstract constraint, but this is a small modification to the method shown here.
In the paper it was shown that information that would normally be lost by an *msg* operator would be preserved by using regular approximations as additional constraints on a set of atoms. The *msg* was still used, and so the use of regular approximations strictly increases precision. Regular approximation was computed only for those terms that were removed during the *msg* operation.
Regular approximations could be integrated with other partial evaluation algorithms such as Conjunction Partial Evaluation [17]. The variables in conjunctions could be constrained by regular predicates in the same fashion.
Staying within the algorithm based on regular approximation, precision can be gained in several ways.

- Nondeterministic RUL programs could be allowed. As seen in Section 2 deterministic RUL programs are less expressive than non-deterministic RUL programs.
- Answer substitutions and other bottom-up computations could be combined with the top-down unfolding algorithm described here. The resulting algorithm would bear a much closer resemblance to a general purpose top-down abstract interpreter, in which both call substitutions and answer substitutions are derived. Assuming a left-to-right computation rule, each iteration of the algorithm collects atoms from resultants only when an answer has already been computed for the atoms to the left in the resultant. The use of regular approximation could increase precision significantly here. Consider a clause body of the form $\leftarrow p(X,Y), q(Y,Z)$ where X is "input", p is some preprocessing stage which passes an intermediate result Y to q. Then a regular approximation of the answers to Y would help to specialise $q(Y,Z)$. Conjunctive partial evaluation is another approach to the problem of maintaining dependencies between atomic goals that share variables.
- Widening could be delayed or applied selectively. Similar considerations apply in abstract interpretation algorithms with widening.
- Regular approximations have been extended with constraints [27], allowing more precise descriptions of sets of terms, such as sorted lists. Arithmetic constraints with the convex hull upper bound, and Boolean constraints are potentially useful constraint domains.

5.1 Implementation

The implementation of the algorithm presented here was based on the SP system [4] for the basic partial evaluation algorithm and the regular approximation tools [6] for the operations concerning the handling of RUL programs. The code runs in SICStus Prolog and is available from the authors.

Acknowledgements

We thank the PEPM'00 referees for their constructive and insightful remarks.

6. REFERENCES

[1] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille.fr/tata, 1999.

[2] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 25–28 June 1995. ACM Press, New York, NY.

[3] D. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation: (LOPSTR-91, Manchester)*. Springer-Verlag Workshops in Computing, 1992.

[4] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[5] J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluati on and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.

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

[7] J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag Lecture Notes in Computer Science, 1996.

[8] P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179 – 210, 1994.

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

[10] N. D. Jones. Combining abstract interpretation and partial evaluation. In P. V. Hentenryck, editor, *Symposium on Static Analysis (SAS'97)*, pages 396–405. Springer Verlag Lecture Notes in Computer Science, Volume 1302, 1997.

[11] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In A. Press, editor, *Conference Record of the Ninth Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[12] L. Lafave. *A Constraint-Based Partial Evaluator for Functional Logic Programs and its Application*. PhD thesis, Department of Computer Science, October 1998.

[13] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, Department of Computer Science, 1997.

[14] M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, *Proceedings of the Symposium on Static Analysis (SAS'98)*, volume 1503, pages 230 – 245. Springer Verlag Lecture Notes in Computer Science, 1998.

[15] M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, pages 220–234, Manchester, UK, June 1998. MIT Press. Extended version as Technical Report CW 259, K.U. Leuven.

[16] 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, *Partial Evaluation*, volume 1110, pages 263 – 283. Springer Verlag Lecture Notes in Computer Science, 1996.

[17] M. Leuschel, D. D. Schreye, and D. A. de Waal. A conceptual embedding of folding into partial deduction: towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*. MIT Press, 1996.

[18] J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[19] P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study*. PhD thesis, Department of Computer Science, May 1999.

[20] P. Mishra. Towards a theory of types in prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, 1984.

[21] T. Æ.. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[22] J. Peralta and J. Gallagher. Imperative program specialisation: An approach using CLP. In A. Bossi, editor, *Pre-Proceedings of Logic Program Synthesis and Transformation (LOPSTR'99)*. University of Venice, 1999.

[23] J. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, pages 246–261. Springer Verlag, Lecture Notes in Computer Science, Volume 1503, 1998.

[24] G. Puebla, M. Hermenegildo, and J. P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Technical report BRICS-NS-99-1, University of Aarhus, pages 75–84, San Antonio, Texas, Jan. 1999.

[25] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, pages 135–151, 1970.

[26] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, The Royal Institute of Technology, 1991.

[27] H. Sağlam and J. Gallagher. Constrained regular approximation of logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation (LOPSTR'97)*. Springer-Verlag, Lecture Notes in Computer Science, 1998.

[28] M. Sørensen and R. Glück. An algorithm of generalisation in positive supercompilation. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming (ILPS'95)*. MIT Press, 1995.

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

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