University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

Regular Trees as an Abstract Domain for Program Specialisation

John P. Gallagher Julio C. Peralta

September 2000 | CSTR-00-010

Regular Trees as an Abstract Domain for Program Specialisation *

John P. Gallagher (john@cs.bris.ac.uk) and Julio C. Peralta †
Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol BS8 1UB, 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 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 subsequent phases of computation cannot be effectively specialised.

In this paper the problem is tackled by introducing regular trees into a partial evaluation algorithm. Regular trees are recursive descriptions of term structure closely related to tree automata. In the algorithm, regular approximations of computation states encountered during partial evaluation are constructed. The critical point is that when generalisation is performed, the upper bound on regular trees can be combined with the *msg*, thus preserving structural information including recursively defined structure. It also allows the specialisation of programs with respect to goals whose arguments range over sets described by regular trees.

The algorithm is presented as an instance of Leuschel's scheme for abstract specialisation of logic programs. This provides a generic algorithm parameterised by an abstract domain - regular trees in this case. The correctness requirements from his scheme are established. The extension of the algorithm to propagate regular approximations of answers as well as calls is also presented, increasing the amount of specialisation that can be obtained. Finally a technique for increasing precision based on "wrapper functions" is introduced.

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*) (Reynolds, 1970) is rather

* This is an extended and revised version of a paper presented at the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'2000), Boston, Mass., January 2000.

† Julio C. Peralta is sponsored by a student grant from DGAPA, UNAM, Mexico.

prone to throw away potential specialisations. A generalisation based on regular approximations, which can preserve much more structural information, is presented here.

The 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 used in off-line partial evaluation, but such an application is not discussed in this paper. Related ideas for binding-time analysis using tree grammars have been used by Mogensen (Mogensen, 1988). Also, regular tree grammars were used by Jones and Muchnick (Jones and Muchnick, 1982) 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. The idea of using regular types to improve generalisation in partial evaluation was also sketched by Leuschel (Leuschel, 1998b).

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 (Gallagher and Lafave, 1996), (Leuschel, 1998a), (Leuschel and Martens, 1996), (Sahlin, 1991; Sørensen and Glück, 1995), (Turchin, 1988).

1.1. Connections Between Abstract Interpretation and Partial Evaluation

Several authors have made the connection between generalisation in partial evaluation and abstract interpretation (de Waal and Gallagher, 1992), (Gallagher, 1993), (Jones, 1997), (Leuschel, 1998b), (Puebla et al., 1999). The key concepts of abstraction and upper bound were introduced in an on-line partial evaluation algorithm in (de Waal and Gallagher, 1992), (Gallagher, 1993) and (Leuschel, 1998b). 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.

Leuschel (Leuschel, 1998b) presented an abstract specialisation algorithm, which was intended to provide an integration of program specialisation techniques with abstract interpretation. The extent to which this goal was achieved can be debated. Leuschel's treatment of concepts from abstract interpretation is non-standard. The abstract domains (which include only downwards-closed domains consisting of elements structured like conjunctions) are not provided with the usual lattice structure. The abstract operations mix concrete and abstract domains, making it impossible to relate the abstract operations and the concrete ones systematically in the usual way, and in fact the concrete semantics is not explicitly defined. Furthermore the "widening" operation does not ensure termination, whereas normally this is the point of widening.

Despite these points, Leuschel's algorithm provides a framework for introducing certain forms of abstraction into partial evaluation, which is very worthwhile and will be adopted for the presentation of the algorithm in Section 3.1.

An alternative approach, adopted in (Puebla et al., 1999) and previously in (Gallagher et al., 1988) is to base an integration on a standard abstract interpretation framework for logic programs based on the operational semantics (such as (Bruynooghe, 1991) or (Muthukumar and Hermenegildo, 1992)). This would provide all the semantic information required to perform specialisation including those specialisations performed by "advanced" techniques such as conjunctive partial evaluation and success information propagation. The emphasis then would be on the invention of suitable domains and widenings for specialisation of programs. However, it is not yet clear how to construct an abstract domain and widenings 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.

An approach based on abstract interpretation promises, in the long run, to unite on-line and off-line partial evaluation, since the "analysis" phase of off-line partial evaluation and the generalisation operation in on-line partial evaluation would both be designed with reference to the structure of a domain of abstract values. Furthermore, abstract interpretation could provide a coherent framework for treating control of unfolding and polyvariance.

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 compiler 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 before the main processing commences.

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 has to be applied to represent the set of results finitely. The generalisation might throw 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 term of form $cont(C, S)$ or $code(C)$ where C is some continuation code and S is a continuation stack. Say the set of possible stacks is $code(c2), cont(c1, code(c2)), cont(c1, cont(c1, code(c2))), \ldots$. An *msg* of all the *cont* terms $cont(c1, X)$. Here X is a variable and destroys any chance of specialising the interpreter, since after some procedure exit the interpreter will continue 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 generalised continuation for the example would be represented as X such that $t(X)$ holds, where t is defined by the following regular logic program.

$$
\begin{array}{l}
t(code(c2)) \leftarrow \\
t(cont(Y, Z)) \leftarrow t1(Y), t(Z) \\
t1(c1) \leftarrow
\end{array}
$$

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

In Section 2, regular trees and their representation as Regular Unary Programs are described. In Section 3, the top-down algorithm is pre-

sented, as an instance of Leuschel's framework. In Section 4, the algorithm is extended to include answer-propagation. In Section 5, a method of increasing precision based on "wrapper functions" is introduced. Sections 6 and 7 contain examples of the application of both algorithms. Section 8 contains conclusions. Appendex A contains the proofs of the conditions required for Leuschel's framework, and Appendix B is the code for the unification example considered in Section 7.

2. Regular Approximations

From now on in this paper the usual terminology and notation of logic programming are adopted (Lloyd, 1987). The concepts are independent of any particular programming language, however. Regular approximation, type graphs, and other grammar-based approximations have been presented in several contexts (Mishra, 1984), (Heintze and Jaffar, 1990), (Janssens and Bruynooghe, 1992), (Van Hentenryck et al., 1994), (Cousot and Cousot, 1995), and closely related ideas can be traced back to (Reynolds, 1969).

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 (Comon et al., 1999); that is they can be used as recognisers for term languages.

An example of an RUL program, and examples of various operations on the program, are displyed in Figure 1.

DEFINITION 1. Success set of a regular unary predicate

Let R be an RUL program and let t be a predicate in R. The *success set* of t in R, denoted $\mathbf{success}_R(t)$ is the set of ground terms u such that $R \cup \{\leftarrow t(u)\}$ has an SLD refutation (that is, $t(u)$ is a consequence of R). The subscript R in **success** may be omitted when the context makes it clear which RUL program is involved.

The success sets of $t1$ and $t2$ in the example program are shown in Figure 1.

Regular unary program P	$t1([]) \leftarrow$ $t1([X\|Y])) \leftarrow s1(X), t1(Y)$ $s1(a) \leftarrow$ $t2([]) \leftarrow$ $t2([X\|Y]) \leftarrow r1(X), t2(Y)$ $r1(b) \leftarrow$
Success set of $t1$ ($\mathbf{success}_P(t1)$)	$\{[], [a], [a,a], \ldots\}$
Success set of $t2$ ($\mathbf{success}_P(t2)$)	$\{[], [b], [b,b], \ldots\}$
True **empty** expression	$\mathbf{empty}((t1([X\|Y]), t2([X\|Y])))$
False **empty** expression	$\mathbf{nonempty}((t1(X)), t2(X)))$
Intersection of $t1$ and $t2$ ($= t3$)	$t3([]) \leftarrow$
Union of $t1$ and $t2$ ($= t4$)	$t4([]) \leftarrow$ $t4([X\|Y])) \leftarrow s1(X), t4(Y)$ $t4([X\|Y])) \leftarrow r1(X), t4(Y)$
Tuple-distributive union of $t1$ and $t2$ ($= t5$)	$t5([]) \leftarrow$ $t5([X\|Y])) \leftarrow s2(X), t5(Y)$ $s2(a) \leftarrow$ $s2(b) \leftarrow$

Figure 1. Operations on Regular Unary Logic Programs

DEFINITION 2. **empty**

Let R be an RUL program, and let $T = t_1(u_1), \ldots, t_m(u_m)$ be a conjunction where $t_1, \ldots, t_m$ are predicates in R and $u_1, \ldots, u_m$ are terms. Then $\mathbf{empty}_R(T)$ is true if there exists a substitution θ such that $u_j\theta \in \mathbf{success}_R(t_j)$, for $1 \leq j \leq m$, and false otherwise. The subscript R in $\mathbf{empty}_R$ may be omitted when the context makes it clear which RUL program is involved. We write $\mathbf{nonempty}(T)$ where $\mathbf{empty}(T)$ is false.

For instance in Figure 1, $\mathbf{empty}((t1([X|Y]), t2([X|Y])))$ is true since there are no instances of $[X|Y]$ (that is, non-nil lists) that are in the success set of both $t1$ and $t2$.

Given any ground atom u, unary predicate t and RUL program R, it is decidable in linear time whether $t(u) \in \mathbf{success}_R(t)$. It is

also decidable in linear time (with respect to the size of R) whether $\mathbf{success}_R(t)$ is empty (Comon et al., 1999). A set of terms is regular (resp. top-down deterministic regular) if and only if it can be defined as the success set of a unary predicate in an RUL (resp. deterministic RUL) program. The special unary predicate *any* has the set of all terms as its success set.

Let u be a term and let $[u]$ represent the set of ground instances of u. It is straightforward to construct a unary predicate t_u and an RUL program R such that $\mathbf{success}_R(t_u)$ is the least regular set containing $[u]$. The definition below allows for a more general case (required in Section 3.6), where the variables of u are constrained by regular predicates. If there are no such constraints on some variable X, the regular predicate *any* is associated with X.

DEFINITION 3. Regular Representation of a Term

Let u be a term with variables $Y_1, \ldots, Y_k$, constrained by predicates $s_1, \ldots, s_k$ respectively. The *regular representation of u with respect to* $s_1, \ldots, s_k$ is given inductively as follows.

$\Leftrightarrow$ The regular representation of a variable Y_j is s_j.

$\Leftrightarrow$ The regular representation of a constant a is a predicate t_a defined by the clause $t_a(a) \leftarrow$.

$\Leftrightarrow$ The regular representation of a term $u = f(u_1, \ldots, u_n)$ is t_u defined by a clause $t_u(f(X_1, \ldots, X_n)) \leftarrow t_{u_1}(X_1), \ldots, t_{u_n}(X_n)$, where $t_{u_1}, \ldots, t_{u_n}$ are the regular representations of $u_1, \ldots, u_n$ respectively.

Note that it is not always possible to represent $[u]$ exactly because u could have repeated variables, as illustrated in the example below.

Example
Let $u = f(X, g(a, X))$. X is not constrained hence the predicate *any* is associated with X. The following predicate t_u is the regular representation of u. Note that, for instance, $f(f(a, a), g(a, a)) \in \mathbf{success}(t_u)$ although it is not an instance of u. The loss of information is due to the repeated variable X.

$t_u(f(X, Y)) \leftarrow any(X), t_g(Y) \qquad t_g(g(X, Y)) \leftarrow t_a(X), any(Y)$
$t_a(a) \leftarrow$

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, then there is an RUL program P' and predicates t_3, t_4 such that $\mathbf{success}_{P'}(t_3) = \mathbf{success}_P(t_1) \cap \mathbf{success}_P(t_2)$ and $\mathbf{success}_{P'}(t_4) = \mathbf{success}_P(t_1) \cup \mathbf{success}_P(t_2)$. Discussions of the operations and their complexity can be found in (Comon et al., 1999). The intersection of two predicates can be constructed in quadratic time (the product of the sizes of of the two programs being intersected) and an additional emptiness check on the result is required (also quadratic).

In Figure 1, the intersection of $\mathbf{success}(t1)$ and $\mathbf{success}(t2)$ is defined by $t3$ and contains only one term []. However, the union of predicates in deterministic RUL programs does not in general result in deterministic RUL programs. The union of $\mathbf{success}(t1)$ and $\mathbf{success}(t2)$ is given by $t4$ in Figure 1. The procedure for $t4$ is non-deterministic since the heads for clauses defining $t4$ have a common instance.

The *tuple-distributive* union of the two predicates is given by predicate $t5$ in Figure 1. The tuple-distributive union is not defined formally here (Mishra, 1984), (Yardeni and Shapiro, 1990). Notice that the success set of $t5$ includes $[a, b]$ which is not contained in the union of $t1$ and $t2$; the tuple-distributive union is the smallest set containing the union that can be represented using a deterministic RUL program. Therefore unlike string languages and conventional finite state automata, in which a non-deterministic automaton can be translated to an equivalent deterministic one, determinacy is a restriction in top-down tree automata. On the other hand the intersection of deterministic predicates is itself deterministic.

2.2. Simplification of Regular Unary Conjunctions

A *canonical conjunction* is a conjunction of unary atoms of the form $r_1(X_1), \ldots, r_k(X_k)$ where $X_1, \ldots, X_k$ are distinct variables. Let R be an RUL program, and $t_1(u_1), \ldots, t_n(u_n)$ be a conjunction of unary atoms with predicates in R. If $\mathbf{empty}((t_1(u_1), \ldots, t_n(u_n))$ is false, there is an equivalent canonical conjunction with the same variables as those occurring in $u_1, \ldots, u_n$.

The equivalent form is derived by first unfolding $t_1(u_1), \ldots, t_m(u_m)$ using the clauses in R, until the conjunction contains no function symbols (that is, the argument of each atom is a variable). Clearly this can be done uniquely due to the form of deterministic RUL programs. Secondly, pairs of atoms having the same argument variable are replaced by their intersection, and this is repeated until the canonical form is

reached. The reduction to canonical form serves as an implementation of the **empty** operation, since the whole conjunction is empty if the intersection of any two atoms with the same argument is empty.

DEFINITION 4. **simplify**

Let R be an RUL program, and $t_1(u_1), \ldots, t_n(u_n)$ be a conjunction of unary atoms with predicates in R. Define $\mathbf{simplify}((t_1(u_1), \ldots, t_n(u_n)))$ to be the canonical form of the conjunction if it exists, or *false* if $\mathbf{empty}((t_1(u_1), \ldots, t_n(u_n)))$ is true.

During the evaluation of **simplify**, any new definitions of predicates arising from intersections are added to R.

Finally, a canonical conjunction can be projected onto a set of variables.

DEFINITION 5. **project**

Let $r_1(X_1), \ldots, r_k(X_k)$ be a canonical conjunction, and let $V \subseteq \{X_1, \ldots, X_k\}$. Then $\mathbf{project}_V((r_1(X_1), \ldots, r_k(X_k)))$ is the conjunction of atoms $r_j(X_j)$ such that $X_j \in V$.

2.3. Widening

The set of all regular sets of terms over a finite signature described by top-down deterministic tree automata 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. Therefore a *widening* (Cousot and Cousot, 1992) is needed in order to ensure convergence of such sequences.

Given a partial order $\sqsubseteq$ and a monotonically increasing sequence $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, a widening operator ∇ is defined as a binary operation satisfying the following conditions.

$\Leftrightarrow$ $\forall x, y : x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$

$\Leftrightarrow$ The increasing sequence defined by $y_0 = x_0, \ldots, y_{i+1} = y_i \nabla x_{i+1}$ is stationary after a finite number of elements. That is, the sequence $y_0, y_1, \ldots$ converges finitely.

Several widenings for regular trees have been suggested (Janssens and Bruynooghe, 1992), (Cousot and Cousot, 1995), (Gallagher and de Waal, 1994), (Van Hentenryck et al., 1994), and their properties analysed (Mildner, 1999). Informally, a widening is employed to introduce recursive definitions into RUL programs. For instance, the sequence of sets

$$\{[]\}, \qquad \{[], [a]\}, \qquad \{[], [a], [a, a]\}, \qquad \ldots$$

is infinite, but a suitable widening operator would force convergence to the set defined by predicate $t6$ below, whose success set contains all finite lists of the element a.

$$t6([]) \leftarrow \qquad\qquad s(a) \leftarrow$$
$$t6([X|Y]) \leftarrow s(X), t6(Y)$$

The idea of widening is that the "growing" list is detected by comparing successive elements of the sequence; a recursive call is introduced at the point where the term is "growing". In our algorithm, 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 t_2 does. An illustration is provided by the representation of the set $\{[], [a], [a, a]\}$.

$$t7([]) \leftarrow \qquad\qquad t8([]) \leftarrow$$
$$t7([X|Y]) \leftarrow s(X), t8(Y) \qquad t8([X|Y]) \leftarrow s(X), t9(Y)$$

$$s(a) \leftarrow \qquad\qquad t9([]) \leftarrow$$

Here $t7$ depends on (that is, calls) $t8$ and the success set of $t8$ is $\{[], [a]\}$ which is contained in the success set of $t7$. Hence, we would form a recursive definition by replacing the occurrence of $t8$ by $t7$ in the body of the clause for $t7$, giving a recursive definition identical to $t6$ shown above (except that $t6$ is renamed $t7$).

This method was used in (Sağlam and Gallagher, 1998) but it is in fact not a true widening as it does not guarantee convergence, as proved by Mildner (Mildner, 1999). However, as noted by Mildner, it appears to converge in all practical cases and could easily be converted to a widening by relaxing the condition that the success set of the predicate is contained in the one on which it depends. For example, this condition could be dropped after some fixed number of iterations of the algorithm. A true widening for type graphs such as the one invented by Van Hentenryck *et al.* (Van Hentenryck et al., 1994) could also be used, though it is computationally more expensive.

Set-based constraints (Heintze and Jaffar, 1990) produce regular approximations (which can be non-deterministic). A widening is not needed in that approach since the structure of the program being analysed statically determines the recursive structure of the approximation. In other words a finite-height, program-specific abstraction is employed for each program to be analysed. Widening-based approximations are potentially more precise than set-based constraints (Cousot and Cousot, 1995), since precision depends not on the program being analysed but on the widening strategy, which can be made as precise

as desired. A widening-based approach seems more appropriate in the context of specialisation because it can be designed to be sensitive to the dynamic aspects of specialisation such as unfolding and polyvariance. However, it would be an interesting line of future research to investigate whether set-based analysis could also be dynamically refined during specialisation.

3. Partial Evaluation With Regular Constraints

3.1. Basic Partial Evaluation Algorithm

A basic partial evaluation algorithm (Gallagher, 1993) takes as input a program and a goal and computes a set of atoms. Each iteration of the program loop applies unfolding (partial evaluation) to the atoms in the current set, yielding a set of *resultants*. The set of atoms in the resultants is *generalised* (to ensure termination) and any new atoms (after generalisation) are added to the current set. Thus the algorithm generates a sequence of sets of atoms on successive iterations, say $A_0, A_1, \ldots$. Let $[A]$ represent the set of instances of A. The sequence is increasing in the sense that $[A_i] \subseteq [A_{i+1}]$, for $i \geq 0$.

Termination occurs when $[A_m] = [A_{m+1}]$ for some m. Thus, the output set of atoms at termination is closed (Lloyd and Shepherdson, 1991), since unfolding the atoms in A_m yields only atoms that are instances of atoms in A_m. 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 ensures termination, since it incorporates some similarity criterion which is used for deciding how many versions of each program procedure appear in the specialised program.

The algorithm in (Gallagher, 1993) 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. In this paper the same style of presentation of the algorithms is followed.

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*.

3.2. Abstract Program Specialisation: Leuschel's Framework

The algorithm with regular approximation will be formulated as an instance of Leuschel's Top-Down Abstract Partial Deduction algorithm (Leuschel, 1998b). Leuschel's algorithm has a similar structure to the basic partial evaluation algorithm of (Gallagher, 1993), but is parameterised by an abstract domain. and abstract operations on that domain. An advantage of adopting Leuschel's algorithm is that it has a correctness proof, which is then applicable to the algorithm developed in this paper after checking the given conditions on the abstract operations. In the following sections, the operations specified by Leuschel will be defined.

3.3. R-Conjunctions and their Representation

The abstract conjunctions of Leuschel's framework are *R-conjunctions*, which are, in effect, conjunctions with some regular constraints on the the variables occurring in them.

DEFINITION 6. R-conjunction
An *R-conjunction* consists of three components:

$\Leftrightarrow$ A conjunction of atoms (say $B_1, \ldots, B_m$);

$\Leftrightarrow$ A canonical conjunction $w_1(X_1), \ldots, w_k(X_k)$, called the *regular constraint* on the conjunction, where $X_1, \ldots, X_k$ are the distinct variables in $B_1, \ldots, B_m$;

$\Leftrightarrow$ A deterministic RUL program R defining $w_1, \ldots, w_k$ (and any of their subsidiary predicates).

An R-conjunction will be written as a triple (B, U, R) where B is the conjunction, U is the regular constraint and R is the RUL program. If an R-conjunction (B, U, R) is such that B is a single atom, it is called an *R-atom*.

The concretisation function γ maps an R-conjunction to the concrete conjunctions that it represents.

DEFINITION 7. γ (concretisation function)
Let (B, U, R) be an R-conjunction. Then $\gamma((B, U, R))$ is the set $\{B\theta \mid \forall\varphi.\mathbf{empty}_R(U\theta\varphi)$ is false$\}$. The function γ is extended to sets of R-conjunctions: $\gamma(S) = \bigcup\{\gamma(A) \mid A \in S\}$.

$\gamma(S)$ is downwards closed as required by Leuschel's framework: that is, $B \in \gamma(S) \Rightarrow \forall\theta.B\theta \in \gamma(S)$. There is a pre-order $\sqsubseteq$ on R-conjunctions given by $A_1 \sqsubseteq A_2$ if and only if $\gamma(A_1) \subseteq \gamma(A_2)$. An equivalence relation $\equiv$ is defined as $A_1 \equiv A_2$ if and only if $A_1 \sqsubseteq A_2$ and $A_2 \sqsubseteq A_1$. The ordering $\sqsubseteq$ induces a partial order on the quotient set of R-conjunctions partitioned according to $\equiv$. Somewhat lazily, we treat $\sqsubseteq$ as a partial order on R-conjunctions rather than dealing with the partial order on the quotient set: this can be justified by picking one representative R-conjunction (given by some standard naming convention for variables and unary predicates) for each equivalence class.

Conceptually each R-conjunction is self-contained. This is, all the RUL clauses associated with an R-conjunction are contained within the triple representing the R-conjunction. This does not imply that in an implementation different R-conjunctions could not share RUL clauses. However, a global RUL store for all R-conjunctions could introduce complications when widening, since widening can apply to some predicate arguments (the ones that are "growing") but not to others. For instance, the same unary predicate might occur within the regular constraints for two different arguments. If a single global representation of that predicate is stored, then any widening which alters the definition of that predicate would automatically apply to both arguments.

3.4. Partial Evaluation Extended with Regular Approximation

The basic algorithm outlined in Section 3.1 is modified to handle R-atoms. Leuschel's framework defines an algorithm, given in Figure 2, which contains two domain-dependent functions, **aunf*** and ω. In turn, **aunf*** requires two subsidiary functions called **aunf** and **ares**. Defining these four functions and checking some associated correctness conditions is sufficient to define a correct abstract specialisation algorithm in Leuschel's framework.

The four operations are now summarised. In the context of this paper, *abstract conjunction* is interpreted as *R-conjunction*. A *resultant* is a formula $B_1 \leftarrow B_2$ where B_1 and B_2 are conjunctions of atomic formulas.

- $\Leftrightarrow$ **aunf**: An abstract unfolding operation that takes an abstract conjunction and returns a set of resultants.

- $\Leftrightarrow$ **ares**: An operation called *abstract resolution* that takes an abstract conjunction (say B) and a resultant (say C) and returns an abstract conjunction.

INPUT: a program P and R-atom A, and operations $\mathbf{aunf}^*, \omega$
OUTPUT: a set of R-atoms

begin
    $S_0 := \{A\}$
    $i := 0$
    repeat
        $S_{i+1} := \omega(\mathbf{aunf}^*(S_i) \cup S_i, S_i)$
        $i := i + 1$
    until $S_i = S_{i-1}$
end

Figure 2. Leuschel's Abstract Top-Down Specialisation Algorithm

$\Leftrightarrow$ $\mathbf{aunf}^*$: An operation $\mathbf{aunf}^*$ on a set of abstract conjunctions, defined using **aunf** and **ares**, introduced mainly for conciseness.

$\Leftrightarrow$ ω: A operation (called a "widening" operator in (Leuschel, 1998b), though it does not conform to the usual definition of widening, as given in Section 2.3). ω takes a set of abstract conjunctions and returns a "widened" set of abstract conjunctions.

The operations are fully explained in Sections 3.5 and 3.6.

3.5. Unfolding R-Atoms

Let A be an R-conjunction (B, U, R), and P be a definite program. To unfold A, the conjunction B is first unfolded finitely in the usual way. That is, a finite SLD-tree for $P \cup \{\leftarrow B\}$ is built (assuming a given unfolding rule). The tree yields resultants $B\theta_1 \leftarrow Q_1, \ldots, B\theta_m \leftarrow Q_m$, where $\leftarrow Q_1, \ldots, \leftarrow Q_m$ are the goals at the leaves of the tree and $\theta_1, \ldots, \theta_m$ are the computed answer substitutions on the respective branches from root to leaf.

The j^{th} resultant $B\theta_j \leftarrow Q_j$ is inconsistent with U if the corresponding instance of the regular constraint, namely $U\theta_j$, is inconsistent, that is, if $\mathbf{empty}_R(U\theta_j)$ is true. In order to define the operations **aunf** and **ares**, the following notation is borrowed from (Leuschel, 1998b). Let P be a definite program, B be a conjunction and τ be a (partial) SLD-tree for $P \cup \{\leftarrow B\}$ (where some unfolding rule determines τ). Denote by $B \Rightarrow_\tau^\theta L$ the fact that $\leftarrow L$ is a goal on a leaf of τ, and θ is the computed answer substitution on the path from the root $\leftarrow B$ to the

leaf $\leftarrow L$. Also, let $C = B_1 \leftarrow B_2$ be a resultant: denote by $Q \Rightarrow_C^\theta L$ the fact that $\theta = mgu(Q, B_1)$ and $L = B_2\theta$.

The operation **aunf** is now defined as follows.

DEFINITION 8. **aunf**

Let A be an R-conjunction (B, U, R). Let P be a program and τ be an SLD-tree for $P \cup \{\leftarrow B\}$. Then $\mathbf{aunf}(A) = \{B\theta \leftarrow Q \mid B \Rightarrow_\tau^\theta Q,\ \mathbf{nonempty}_R(U\theta)\}$.

Example

Let an R-atom be (B, U, R) where $B = p(X, Y)$, $U = t1(X), t2(Y)$ and R is the following RUL program.

$$\begin{array}{ll} t1(a) \leftarrow & t2(b) \leftarrow \\ t1(f(X)) \leftarrow t1(X) & t2(f(X)) \leftarrow t2(X) \end{array}$$

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

$$p(Z, Z) \leftarrow q(Z) \qquad p(f(X_1), f(Y_1)) \leftarrow p(X_1, Y_1)$$

The corresponding substitutions $\{X/Z, Y/Z\}$ and $\{X/f(X_1), Y/f(Y_1)\}$ are applied to the regular constraint $t1(X), t2(Y)$, giving the constraints $t1(Z), t2(Z)$ and $t1(f(X_1)), t2(f(Y_1))$. The first of these is inconsistent as $\mathbf{empty}((t1(Z), t2(Z)))$ is true. The second resultant is consistent since $\mathbf{simplify}((t1(f(X_1)), t2(f(X_1))))$ equals $t1(X_1), t2(Y_1)$ and this has a solution (such as $\{X_1/a, Y_1/b\}$). Hence only the second resultant would be returned and $\mathbf{aunf}((B, U, R)) = \{p(f(X_1), f(Y_1)) \leftarrow p(X_1, Y_1)\}$.

As prescribed by Leuschel's framework, an *abstract resolution* operation **ares** is now defined.

DEFINITION 9. **ares**

Let A be an R-conjunction (B, U, R). Let C be a resultant. Let τ be an SLD-tree for $P \cup \{\leftarrow B\}$. Then $\mathbf{ares}(A, C)$ is the R-conjunction (Q, U', R'), where $B \Rightarrow_C^\theta Q$, $U' = \mathbf{project}_{vars(Q)}(\mathbf{simplify}(U\theta))$ and $\mathbf{nonempty}_R(U\theta)$. (Leuschel's framework does not specify the value of $\mathbf{ares}(A, C)$ if there is no Q, θ such that $B \Rightarrow_C^\theta Q$, since their existence is guaranteed wherever **ares** is evaluated). R' is the RUL program obtained by adding to R the clauses defining intersections computed during **simplify**, if any.

Continuing the previous example, $\mathbf{ares}((B, U, R), p(f(X_1), f(Y_1)) \leftarrow p(X_1, Y_1)) = (p(X_1, Y_1), (t1(X_1), t2(Y_1)), R)$.

The function $\mathbf{aunf}^*$ is from Leuschel's framework. $\mathbf{aunf}^*$ takes a set of R-atoms and returns a set of R-atoms.

DEFINITION 10. **aunf***

Let S be a set of R-atoms. Define $\mathbf{aunf}^*(S) = \{\mathbf{ares}(A, C) \mid A \in S, C \in \mathbf{aunf}(A)\}$.

3.6. Upper Bound of R-Atoms

The next requirement for Leuschel's algorithm is to construct a generalisation operator, called ω in (Leuschel, 1998b). The construction of ω is described next. The most important component of ω in the domain of R-conjunctions is an operation to compute the upper bound of R-conjunctions.

Let A_1, A_2 be R-conjunctions (B_1, U_1, R_1) and (B_2, U_2, R_2) where (B_1, U_1) is renamed apart so that it has no variable in common with (B_2, U_2). Assume that the upper bound is computed only between two R-conjunctions where B_1 and B_2 have the same sequence of predicates. Let B be an *msg* of B_1 and B_2, (and hence B also has the same sequence of predicates), such that B contains no variable in common with B_1 and B_2. Then there are substitutions θ_1 and θ_2 such that $B\theta_1 = B_1$ and $B\theta_2 = B_2$. Furthermore every variable in B is bound in both θ_1 and θ_2.

For each variable X in B, a regular constraint $t_X(X)$ is constructed as follows. Let X/r_1 and X/r_2 be X's bindings in θ_1 and θ_2 respectively. r_1 and r_2 are subterms of B_1 and B_2 respectively, possibly containing variables whose values are described by unary predicates in U_1 and U_2. Let t_{r_1} and t_{r_2} be the regular representations of r_1 and r_2 with respect to $\mathbf{project}_{vars(r_1)}(U_1)$ and $\mathbf{project}_{vars(r_2)}(U_2)$ respectively (see Definition 3). Then t_X is defined to be the tuple-distributive union of t_{r_1} and t_{r_2}. Thus, t_X records a regular approximation of the values to which X could be bound in the terms of which X is a generalisation. Using *msg* alone, this information would be lost.

Example

First a trivial example is shown to give the basic idea. Let two R-atoms be $(p(a), true, \emptyset)$ and $(p(a), true, \emptyset)$.

Then their upper bound is given by the R-atom $(p(X), t(X), R)$ where R is the RUL program $\{t(a) \leftarrow,\ \ t(b) \leftarrow\}$.

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

Example

Let two R-atoms be (B_1, U_1, R_1) and (B_2, U_2, R_2), where

$\Leftrightarrow\ B_1 = p(f(a), X, a),\ U_1 = t1(X);$

$\Leftrightarrow$ $B_2 = p(f(f(a)), Z, f(a))$, $U_2 = t3(Z)$;

$\Leftrightarrow$ R_1 and R_2 are defined as follows.

R_1	R_2
$t1([]) \leftarrow$	$t3([]) \leftarrow$
$t1([X\|Xs]) \leftarrow t2(X), t1(Xs)$	$t3([X\|Xs]) \leftarrow t4(X), t3(Xs)$
$t2(a) \leftarrow$	$t4(b) \leftarrow$

The *msg* of $p(f(a), X, a)$ and $p(f(f(a)), Z, f(a))$ is $p(f(Y), W, Y)$. The substitutions on the *msg* to obtain the original atoms are $\{Y/a, W/X\}$ and $\{Y/f(a), W/Z\}$. The union of the regular representations of the terms bound to Y (that is, a and $f(a)$) is given by predicate $t5$.

$$t5(a) \leftarrow \qquad t5(f(X)) \leftarrow t6(X)$$
$$t6(a) \leftarrow$$

The upper bound of the terms bound to W is computed by finding the tuple-distributive union of the predicates $t1$ and $t3$, which is $t7$ below. This gives the final R-atom (B, U, R) where $B = p(f(Y), W, Y)$, $U = t5(Y), t7(W)$ and R is defined as follows.

$t5(a) \leftarrow$	$t7([]) \leftarrow$
$t5(f(X)) \leftarrow t6(X)$	$t7([X\|Xs]) \leftarrow t8(X), t7(Xs)$
$t6(a) \leftarrow$	$t8(a) \leftarrow$
	$t8(b) \leftarrow$

The example shows that, although the tuple-distributive union discards some argument dependencies, the *msg* retains some dependency information. 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.

DEFINITION 11. **atoms**

Let $A = (B, U, R)$ be an R-conjunction. The operation $\mathbf{atoms}(A)$ returns a set of R-atoms from A. Suppose $B = B_1, \ldots, B_m$. Then $\mathbf{atoms}(A) = \{(B_j, \mathbf{project}_{vars(B_j)}(U), R) \mid 1 \leq j \leq m\}$. That is, each atom in B forms an R-atom, with the appropriate regular constraints projected from U. **atoms** can be extended to a set of R-conjunctions: $\mathbf{atoms}(S) = \bigcup\{\mathbf{atoms}(A) \mid A \in S\}$.

atoms(S) is an *abstraction* of S according to the definition in (Leuschel, 1998b). That is, for every R-conjunction A in S there exists a sequence of R-atoms $\langle A_1, \ldots, A_m \rangle$ from **atoms**(S) such that the following condition holds.

$$\{(Q_1, \ldots, Q_m) \mid \forall i : 1 \leq i \leq m \;.\; Q_i \in \gamma(A_i)\} \subseteq \gamma(A)$$

An operation called **partition** is introduced to control the number of versions of each predicate. Let S be a set of R-atoms. Form a finite number of disjoint non-empty partitions of S, where each partition consists of "similar" R-atoms and within each partition the R-atoms have the same predicate. Suppose the partition of S is $\{S_1, \ldots, S_k\}$; then define **partition**$(S) = \{A_1, \ldots, A_k\}$ where A_j is the upper bound of the R-atoms in S_j, $1 \leq j \leq k$.

The operation ω in the algorithm in Figure 1 is defined as follows.

DEFINITION 12. ω

Let S, S' be sets of R-conjunctions: then define the generalisation operation $\omega(S, S') = S' \nabla \textbf{partition}(\textbf{atoms}(S))$ where ∇ is a widening operator.

In the algorithm in Figure 2, the first argument S' of ∇ is the set of R-atoms constructed during the previous iteration of the algorithm. Although the ω operation in (Leuschel, 1998b) has only one argument, a footnote in the paper mentions the possibility of adding extra arguments, depending on the widening employed.

3.7. Generation of the Specialised Program

A specialised program is obtained from the final set of R-atoms generated by the algorithm, say S_n. The specialised program consists of the set of resultants **rename**$(\bigcup\{\textbf{aunf}(A) \mid A \in S_n\})$, where **rename** is an operation for renaming predicates, using the technique used in other specialisers (described in (Gallagher, 1993) and (Leuschel, 1998b) among others). We do not rename the predicate for the initial goal, so as to preserve the external interface of the specialised program.

PROPOSITION 13. *Let P be a definite program and G be an R-atom. Let P' be the program obtained from the final set of R-atoms computed by the algorithm in Figure 2. Then for all $Q \in \gamma(G)$, $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer substitution θ if and only if $P' \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer substitution θ.*

The generic proof is given by Leuschel in (Leuschel, 1997b). The proofs assume certain conditions on the operations **aunf**, **ares** and ω. Proofs of these conditions are given in Appendix A.

4. Bottom-Up Propagation of Answers

In the partial evaluation algorithm discussed above, the information contained in the arguments of the initial call is propagated top-down by unfolding. As is known, this technique has limitations, but it can be augmented by bottom-up propagation of information from within the program. In this way it is possible to specialise procedures within the program whose input comes from intermediate data structures. Consider the following very simple example which illustrates the essential point.

$$
\begin{array}{ll}
p \leftarrow input(X), output(X) & output([]) \leftarrow \\
input([]) \leftarrow & output([a|Xs]) \leftarrow output(Xs) \\
input([a|X]) \leftarrow input(X) & output([b|Xs]) \leftarrow output(Xs)
\end{array}
$$

The input X ranges over lists containing a. This implies that the *output* procedure should be specialised to handle lists of a, pruning the clause $output([b|Xs]) \leftarrow output(Xs)$ which cannot possibly be used. But the specialisation of *output* cannot be achieved by the basic algorithm, which records only calls to procedures. This is because whatever unfolding rule is used, at least one computation branch will contain a call to *output* with an open-ended list as its argument. Hence the *output* procedure cannot be specialised.

In this section regular approximation will be used to obtain an approximation to the set of all answers to $input(X)$. The *output* procedure is then specialised with respect to that approximation. Propagation of answers in partial evaluation has been proposed before (Leuschel and De Schreye, 1996), (Vanhoof et al., 1998) but the use of *msg* as a generalisation would not achieve the required specialisation in the example above. Regular approximation provides a more precise generalisation for answers, as was the case for calls.

This particular simple example above could be handled in other ways, for example by unfold-fold transformation (Tamaki and Sato, 1984b) or by Conjunctive Partial Evaluation (De Schreye et al., 1999). However, the method of bottom-up propagation handles more complex examples. Consider inserting code between *input* and *output*. In the example below the input list is reversed before calling *output*.

$$
\begin{array}{l}
p \leftarrow input(X), reverse(X,Y), output(Y) \\
reverse(X,Y) \leftarrow rev1(X,Y,[]) \\
rev1([],Z,Z) \leftarrow \\
rev1([X|Xs],Y,Z) \leftarrow rev1(Xs,Y,[X|Z])
\end{array}
$$

(*Definitions of input and output as before*)

INPUT: a program P and atomic goal G
OUTPUT: two sets of R-atoms (calls and answers)

begin
    $A :=$ the R-atom corresponding to G
    $S_0 := \{A\}$
    $T_0 := \{\}$
    $i := 0$
    repeat
        $S_{i+1} := S_i \nabla \mathbf{partition}(\mathbf{calls}(S_i \cup \mathbf{aunf}^*(S_i), T_i))$
        $T_{i+1} := T_i \nabla \mathbf{answers}(S_i, T_i)$
        $i := i + 1$
    until $S_i = S_{i-1}$ and $T_i = T_{i-1}$
end

Figure 3. Partial Evaluation Algorithm with Answer Propagation

It seems that the other techniques mentioned above would not be able to specialise *output* in this program. The extended algorithm shown in the next section obtains the result that the output of $reverse(X, Y)$ is a list of a and hence $output(Y)$ can be specialised.

4.1. Basic Algorithm Extended with Answer Propagation

The main change to the algorithm in Figure 2 is the introduction of a second set of R-atoms, to represent answers. In Figure 3, the sets S_i contain approximations of call atoms extracted from the resultants. The set T_i contains approximations of answers to the atoms in S_i. That is, for each call generated during unfolding, the set of answers for that call is recorded and stored. Obviously the set of answers for a given call could be infinite and so some generalisation operation is needed. Regular approximation provides a suitably precise operation.

The introduction of answers implies a change to the way calls are generated. Consider a conjunction $B_1, \ldots, B_m$, generated during partial evaluation. Assuming a left-to-right propagation of information in clause bodies, atom B_j will be called only after computing answers for the atoms to its left, namely $B_1, \ldots, B_{j-1}$. Suppose φ is an answer substitution for $B_1, \ldots, B_{j-1}$. Then $B_j\varphi$ is a call. If at least one of $B_1, \ldots, B_{j-1}$ fails or loops then B_j is never called.

4.2. Representing Answers as R-atoms

Answers for R-atoms are also represented as R-atoms. Let (B, U, R) be an R-atom. An R-atom (B_1, U_1, R_1) is an *answer for* (B, U, R) if B_1 is an instance of B, say $B\theta = B_1$, and $\mathbf{empty}((U\theta, U_1))$ is false.

DEFINITION 14. Answer Constraint for an R-conjunction

Let (B, U, R) be an R-conjunction. Given a set of R-atoms T, an *answer constraint* for (B, U, R) in T (if such an answer exists) is constructed inductively as follows. Note that the answer constraint is a pair consisting of a substitution and a canonical conjunction.

1. If B is a single atom, and $(B_1, U_1, R_1) \in T$ is an answer for (B, U, R) then return $(\theta, \mathbf{simplify}((U\theta, U_1)))$, where $B\theta = B_1$.

2. If $B = A_1, \ldots, A_n$, $n > 1$, and $(B_1, U_1, R_1) \in T$ is an answer for $(A_1, \mathbf{project}_{vars(A_1)}(U), R)$, and $A_1\theta = B_1$, then return the answer constraint for (B_2, U_2, R_2) where $B_2 = (A_2, \ldots, A_n)\theta$, $U_2 = \mathbf{simplify}((U\theta, U_1))$, and R_2 is obtained by adding to R the definitions of new predicates obtained when evaluating **simplify**.

The operations in Figure 3 include $\mathbf{aunf}^*$, **partition**, and the widening ∇, which were used in the algorithm in Figure 2. The operation $\mathbf{calls}(S,T)$ is a refinement of the operation $\mathbf{atoms}(S)$ of Figure 2, returning only the R-atoms whose predecessors to the left have answers in T.

DEFINITION 15. **calls**

Let $A = (B, U, R)$ be an R-conjunction and let T be a set of R-atoms. Let $B = B_1, \ldots, B_n$. Define $\mathbf{calls}(A, T)$ as follows.

$$\begin{aligned}\mathbf{calls}(A, T) = \{(B_j,\ & \mathbf{project}_{vars(B_j)}(\mathbf{simplify}((U_1, U))), R_1) \mid \\ & U_1 \text{ is an answer constraint for } B_1, \ldots, B_{j-1} \text{ in } T, \\ & 1 \leq j \leq n\}\end{aligned}$$

calls is extended so that its first argument is a set of R-conjunctions in the usual way.

Next, consider the construction of answers. Given an R-atom A and a set of answers T, answers for A can be constructed by unfolding A and answering the resulting unfolded conjunctions. The function **ans** encapsulates this.

DEFINITION 16. **ans**

Let $A = (B, U, R)$ be an R-atom, and T be a set of R-atoms. Define $\mathbf{ans}(A, T)$ as follows.

$$\begin{aligned}\mathbf{ans}(A,T) = \{(B\theta,\ & \mathbf{project}_{vars(B\theta)}(\mathbf{simplify}((U\theta, U_1)), R_1) \mid \\ & A_1 \in \{\mathbf{ares}(A, C) \mid C \in \mathbf{aunf}(A)\}, \\ & A_1 \text{ has answer constraint } (\theta, U_1) \text{ in } T\}\end{aligned}$$

R_1 is obtained from R by adding the definitions of new predicates produced while evaluating the answer for A_1.

DEFINITION 17. **answers**
Let S and T be sets of R-atoms.
Define $\mathbf{answers}(S, T) = \bigcup\{\mathbf{ans}(A, T) \mid A \in S\}$.

As with the basic algorithm in Figure 2, the algorithm in Figure 3 could be optimised to take advantage of those calls and answers newly added on the previous iteration.

The residual code is generated from the final sets S and T

$$\begin{aligned}\{\mathbf{rename}(H\theta\rho \leftarrow B\rho \mid\ & (H, U, R) \in S, \\ & H\theta \leftarrow B \in \mathbf{aunf}((H, U, R)) \\ & (\rho, U') \text{ is an answer constraint for } B \text{ in } T\}\end{aligned}$$

The sketch of a proof for the correctness of the algorithm is as follows. Upon termination, the set S is closed with respect to the final set of answers T. That is, unfolding an R-atom in S and applying the answers from T to the resultants, gives a set of calls that are subsumed by R-atoms already in S. Similarly, T is such that $\mathbf{answers}(S, T)$ contains only answers subsumed by R-atoms in T. It would then be argued that the set of calls produced by any atom in $\gamma(S)$ (using the same unfolding rule as was used for atoms in S) are represented by atoms in $\gamma(S)$. An inductive proof would then establish that any refutation of a concrete atom represented by S would be reproduced by a finite derivation using the resultants in the residual program.

4.3. Comparison with Leuschel's Top-Down/Bottom-Up Algorithm

Leuschel (Leuschel, 1998b) also developed an algorithm incorporating answer-propagation, as an extension of his abstract specialisation algorithm discussed earlier. Though some of the ideas are the same, we have not presented our answer propagation algorithm with regular approximation as an instance of his framework, as we did with the top-down algorithm. The reasons are that there are significant differences in the operation of his algorithm.

1. Answers are not propagated from left to right in his algorithm. The notion of *refinement* in his framework, which takes the answers into account, appears to allow *all* possible ways of inserting answers, at any point in a resultant. Perhaps the intention of the framework is to fix some particular order for a specific instance of the algorithm, but this is not clear. As it stands his framework allows calls to be generated before it is known whether they can actually be called in the standard left-to-right execution. This also implies that specialisations are missed.

2. Our answer propagation preserves variable dependencies between R-atoms, whereas in Leuschel's framework the effect of answering an abstract conjunction is limited to that conjunction, and not propagated to the rest of the resultant of which it is a part. The fact that Leuschel uses conjunctions rather than atoms as the basic unit to be unfolded does preserve some variable dependencies, since dependencies within conjunctions are preserved.

3. Leuschel's algorithm itself is not a global fixpoint computation but uses a fixpoint on top-down execution *within* each outer loop iteration. This does not seem to be necessary, since methods such as OLDT (Tamaki and Sato, 1986), as well as abstract interpretation frameworks such as (Bruynooghe, 1991) and (Muthukumar and Hermenegildo, 1992) have long established that a global fixpoint computation is complete for combining top-down and bottom-up execution. Our algorithm does in fact resemble OLDT and related techniques. It seems that Leuschel's claim that OLDT could be reconstructed in "a slight extension" of his framework is optimistic.

5. Increased Precision Using Wrapper Functions

It was mentioned in Section 2 that the success set of the upper bound of RUL predicates t and s is a superset of the union of the success sets of t and s. This leads to a loss of precision that can sometimes prevent desirable specialisations. Loss of precision is inevitable of course but some control over the loss is possible, while remaining in the domain of deterministic RUL programs, by using *wrapper functions*.

To introduce the concept of wrapper functions, consider the upper bound of R-atoms $(p(a,b), true, \emptyset)$ and $(p(c,d), true, \emptyset)$. The tuple-distributive upper bound of the two R-atoms, which will include $p(a,d)$ and $p(c,b)$ among the atoms represented, is $(p(X,Y), (t5(X), t6(Y)), R)$

where R is as follows.

$$t5(a) \leftarrow \qquad t6(b) \leftarrow$$
$$t5(c) \leftarrow \qquad t6(d) \leftarrow$$

The idea of wrapper functions is to "wrap" any tuple of arguments which is to be kept distinct in a unique function symbol. Suppose that in the example above $p(a, b)$ and $p(c, d)$ are not to be confused with each other. Then the wrapper functions $f1$ and $f2$ are introduced, changing the atoms to $p(f1(a, b))$ and $p(f2(c, d))$ respectively. Computing the tuple-distributive upper bound of the corresponding two R-atoms now gives precisely the union, represented as $(p(X), s0(X), R)$, with R defined below.

$$s0(f1(X, Y)) \leftarrow s2(X), s3(Y) \qquad s0(f2(X, Y)) \leftarrow s5(X), s6(Y)$$
$$s2(a) \leftarrow \qquad s5(c) \leftarrow$$
$$s3(b) \leftarrow \qquad s6(d) \leftarrow$$

It is clear that the tuples $f1(c, b)$ and $f1(a, d)$ are not included in the upper bound. Naturally the introduction of the wrapper functions in a procedure requires changes to the rest of the program (that is, any procedures that call $p(X)$ in the above example will have to "peel off" the wrappers). Although wrapper functions represent *ad hoc* control for a particular program, we have experimented with the systematic introduction of wrappers. In the example in Section 6 it will be shown how wrapper functions can enable a precise representation of the stack of activation records in an interpreter for an imperative language. A simpler example is given in this section to illustrate the combination of wrapper functions with widening.

5.1. Wrapper Functions: Example

Consider a graph consisting of edges (a, b), (b, c), (b, d), (b, e), and (c, a). A program to represent the graph, and paths in the graph, is shown.

$$edge(a, b) \leftarrow \qquad edge(b, c) \leftarrow$$
$$edge(b, d) \leftarrow \qquad edge(d, e) \leftarrow$$
$$edge(c, a) \leftarrow$$

$$path(X, Y, [(X, Y)]) \leftarrow edge(X, Y)$$
$$path(X, Z, [(X, Y)|P]) \leftarrow edge(X, Y), path(Y, Z, P)$$

The third argument of *path* is a list of edges encountered on the path. Consider specialising the program with respect to $path(a, X, Y)$ (a query to find a path starting at a). Since there is a cycle in the graph there is an infinite set of paths. A recursive approximation of

the third argument of *path* gives somewhat inaccurate information. Tuple-distributivity among the edges introduces spurious edges such as (b, a), (b, b), (b, e) and so on. Changing the representation using wrappers results in a program such as the following. Notice that the path now contains wrapped edges. For instance, where the original program might have contained a list element (b, c) the revised program will have $e2(b, c)$.

$$\begin{array}{ll} wrapped\_edge(e1(a,b),a,b) \leftarrow & wrapped\_edge(e2(b,c),b,c) \leftarrow \\ wrapped\_edge(e3(b,d),b,d) \leftarrow & wrapped\_edge(e4(d,e),d,e) \leftarrow \\ wrapped\_edge(e5(c,a),c,a) \leftarrow & \end{array}$$

$$\begin{array}{l} path(X,Y,[E]) \leftarrow wrapped\_edge(E,X,Y) \\ path(X,Z,[E|P]) \leftarrow wrapped\_edge(E,X,Y), path(Y,Z,P) \end{array}$$

Specialisation with respect to $path(a, X, Y)$ now gives an approximation of the paths which contains only valid edges since the wrappers $e1, e2$ and so on ensure that the edges are not confused during generalisation.

The advantage of the extra precision given by the wrapped edges comes not from specialising the program given above, but rather in specialising a program that uses the path. Suppose the *path* procedure were used in a context of the form:

$$path(a, X, P), processPath(P).$$

Then the procedure $processPath$ would be revised to exploit the wrapper information. Wherever the original procedure contains a clause of the form $processPath([(X, Y)|P]) \leftarrow Body$, the revised program would contain a clause for each wrapper function.

$$\begin{array}{l} processPath([e1(X,Y)|P]) \leftarrow Body \\ processPath([e2(X,Y)|P]) \leftarrow Body \\ etc. \end{array}$$

The repetition of the procedure for each wrapper function allows the specialiser to recover the actual edges via the wrapper. The specialised procedure for $processPath$ will have a precise case for each edge that can occur in the path. Probably the usefulness of this approach is reduced in programs where the number of wrappers leads to a large increase in code size due to the replication of code as shown.

We observe that the use of wrappers appears to be the opposite strategy to the *untupling* strategy used in (Codish et al., 2000) for increasing precision in analyses, introducing extra structure instead of removing it. Untupling is a program transformation that promotes

terms where possible to the level of argument of predicates. Greater precision can sometimes be obtained by analysing the transformed program. In general terms the aims of untupling are the same as wrapper functions - namely, to distinguish terms that might otherwise be confused by the operations on the abstract domain. However, the purpose of wrappers is to distinguish terms that may appear at arbitrary depth in an argument and hence cannot be promoted by the untupling transformation. Untupling would have no effect on the specialisation of the *processPath* procedure above.

6. Example: Specialisation of Operational Semantics

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 (Peralta et al., 1998) using a variable environment, the program abstract syntax tree, and a stack of activation records. The language is intentionally 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. To reduce the complexity of analysis the interpreter is "one-state" and does not explicitly return a "final" state. For more discussion of the semantics see (Peralta et al., 1998), (Peralta and Gallagher, 2000). In (Peralta, 2000) a more complex language is treated. 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. In Figure 4, the logic program defining the semantics [3] of our imperative language is presented.

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 formal parameters is passed to the actuals and variable scope is updated accordingly. The clauses in Figure 4 include some of the predicates needed for specialisation of a *factorial* program.

[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.

$statement(Env, [], fin) \leftarrow$
$statement(Env, [skip|Prg], St) \leftarrow$
$\quad statement(Env, Prg, St)$
$statement(Env, [assign(X, Expr)|Prg], St) \leftarrow$
$\quad a\_expr(Env, Expr, Vl),$
$\quad update(Env, Env1, X, Vl),$
$\quad statement(Env1, Prg, St)$
$statement(Env, [if(B\_test, S1, S2)|Prg], St) \leftarrow$
$\quad b\_expr(Env, B\_test, true),$
$\quad compose(S1, Prg, SPrg),$
$\quad statement(Env, SPrg, St)$
$statement(Env, [if(B\_test, S1, S2)|Prg], St) \leftarrow$
$\quad b\_expr(Env, B\_test, false),$
$\quad compose(S2, Prg, SPrg),$
$\quad statement(Env, SPrg, St)$
$statement(Env, [call(Prc\_n, Arg)|Prg], St) \leftarrow$
$\quad address(Prg, Ad),$
$\quad code(Prc\_n, proc(Actls, Prc\_b)),$
$\quad a\_expr(Env, Arg, Vl),$
$\quad vars(Env, Env1, Actls, Vl, R),$
$\quad statement(Env1, Prc\_b, fr(R, Ad, St))$
$statement(Env, [rtn|\_], fr(R, Ad, St)) \leftarrow$
$\quad restore(Env, Env1, R),$
$\quad code\_id\_cont(Env1, Ad, St)$

$address([], 1) \leftarrow$
$address([assign(f, times(f, a)), rtn], 2) \leftarrow$
$code(fac, proc(a,$
$\quad [if(gt(n, 0), [assign(n, minus(n, 1)),$
$\quad\quad call(fac, n),$
$\quad\quad assign(f, times(f, a))],$
$\quad\quad [skip]),$
$\quad rtn])) \leftarrow$

$code\_id\_cont(N, 1, TrP) \leftarrow$
$\quad statement(N, [], TrP)$
$code\_id\_cont(N, 2, TrP) \leftarrow$
$\quad statement(N, [assign(f, times(f, a)), rtn], TrP)$
$code\_id\_cont(N, 3, TrP) \leftarrow$
$\quad statement(N, [assign(f, plus(f, a))], TrP)$

...(possibly other continuation definitions)

Figure 4. Semantics-Based Interpreter for an Imperative Language

$$
\begin{aligned}
&statement(X1) \leftarrow \\
&\qquad statement\_2(X1, X2, 1, fin) \\
&statement\_2(X1, X2, X3, X4) \leftarrow \\
&\qquad X1 > 0, \\
&\qquad statement\_3(X1, X2, X3, X4) \\
&statement\_2(X1, X2, X3, X4) \leftarrow \\
&\qquad X1 =< 0, \\
&\qquad statement\_4(X1, 1, X1, fr(a(X2), X3, X4)) \\
&statement\_3(X1, X2, X3, X4) \leftarrow \\
&\qquad X5 is X1 \Leftrightarrow 1, \\
&\qquad statement\_5(X5, X1, X2, X3, X4) \\
&statement\_4(X1, X2, X3, fr(a(X4), X5, X6)) \leftarrow \\
&\qquad code\_id\_cont\_8(X1, X2, X4, X5, X6) \\
&statement\_5(X1, X2, X3, X4, X5) \leftarrow \\
&\qquad statement\_2(X1, X2, 2, fr(a(X3), X4, X5)) \\
&statement\_7(X1, X2, X3, X4) \leftarrow \\
&\qquad X5 is X2 * X3, \\
&\qquad statement\_4(X1, X5, X3, X4) \\
&code\_id\_cont\_8(X1, X2, X3, 1, fin) \leftarrow \\
&code\_id\_cont\_8(X1, X2, X3, 2, X4) \leftarrow \\
&\qquad statement\_7(X1, X2, X3, X4)
\end{aligned}
$$

Figure 5. Result of Specialisation of Interpreter w.r.t. *factorial*

Now specialise the interpreter with respect to a procedure call (factorial) with input environment having three variables, namely n, f and a where n and a are unknown and f is 1; fin is the initial value of the activation record stack. Such information is encoded into the query $\leftarrow statement([\_, \_, 1], [p(1, call(fac, n))], fin)$. After specialisation with respect to the query above, the residual program in Figure 5 is obtained.

Existing specialisers (Leuschel, 1997a),(Sahlin, 1991) 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 Figure 5 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 *msg* generalisation include such redundant code. In this case regular approximations prevent information loss during generalisation.

6.1. Using Wrapper Functions in the Operational Semantics

The *address* procedure was used above to record the various pieces of code that could be pushed onto the stack of activation records. It would require some non-trivial analysis of the source code to produce this automatically. In this section wrapper functions are used instead. The effect is the same, namely, to produce a sufficiently precise representation of the stack so that the specialiser can determine which code to execute on return from a procedure. Recall that the same procedure could be called from several places in the code, so that different continuations might apply to the same procedure return. A more comprehensive discussion of introducing wrapper functions into the code of semantics-based interpreters is contained in (Peralta, 2000).

Consider the clause that handles a procedure call.

$$\begin{aligned} &statement(Env, [call(Prc\_n, Arg)|Prg], St) \leftarrow \\ &\quad address(Prg, Ad), \\ &\quad code(Prc\_n, proc(Actls, Prc\_b)), \\ &\quad a\_expr(Env, Arg, Vl), \\ &\quad vars(Env, Env1, Actls, Vl, R), \\ &\quad statement(Env1, Prc\_b, fr(R, Ad, St)) \end{aligned}$$

The *call* statements of the imperative source program are labelled with unique *program points*. Suppose each procedure call statement of the form $call(Prc\_n, Arg)$ is replaced by a term $p(L, call(Prc\_n, Arg))$ where L is the unique label identifying this statement. The frame to be pushed onto the stack is the continuation code Prg and the current variable values R are to be restored upon return from the procedure.

In place of the *address* procedure is a procedure linking program points and wrapper functions. For each program point L introduce a clause

$$wrapper(L, X, fL(X)) \leftarrow$$

where fL is a function symbol unique to L. The clause above is reformulated as follows.

$$\begin{aligned} &statement(Env, [p(L, call(Prc\_n, Arg))|Prg], St) \leftarrow \\ &\quad code(Prc\_n, proc(Actls, Prc\_b)), \\ &\quad a\_expr(Env, Arg, Vl), \\ &\quad vars(Env, Env1, Actls, Vl, R), \\ &\quad wrapper(L, (R, Prg), F), \\ &\quad statement(Env1, Prc\_b, fr(F, St)) \end{aligned}$$

The frame pushed onto the stack in the final line of the clause is of the form $fL((R, Prg))$ and hence the values of R and Prg will not

be confused with any other values obtained from other program points during generalisation.

Finally, when execution returns from the procedure and a frame is popped from the stack, a procedure unwraps the stack frames. The clause for procedure return in the semantics is rewritten.

$$statement(Env, [rtn|\_], fr(F, St)) \leftarrow \\ code\_id\_cont(F, Env, St)$$

The procedure *code_id_cont* performs the unwrapping by matching all possible wrapper functions, with a clause of the following form for each wrapper fL.

$$code\_id\_cont(fL((R, Prg)), Env, St) \leftarrow \\ restore(Env, Env1, R), \\ statement(Env1, Prg, St)$$

This technique has been adapted in (Peralta, 2000) for interpreters for various imperative languages. The procedures for *wrapper* and *code_id_cont* are specific to a given object program, but no analysis is required to produce them: the list of program points is all that is required.

7. Examples of Answer-Propagation During Specialisation

7.1. Specialisation of Unification

Specialisation of the unification algorithm is relevant in contexts such as compilation of logic programs (Kursawe, 1987), (Gurr, 1994) and resolution theorem provers. The problem of specialising unification was discussed in (de Waal and Gallagher, 1992). The unification program given in full in Appendix B handles terms in the ground representation, in which terms are either of the form $var(N)$ where N is a number, or $struct(F, Args)$ where F is a function symbol and $Args$ is a list of its arguments. $unify(X, Y, S)$ means that terms X and Y have *mgu* S.

A *ground* term in this representation can be described by a regular constraint.

$$ground(struct(X1, X2)) \leftarrow any(X1), groundArgs(X2)$$
$$groundArgs([]) \leftarrow$$
$$groundArgs([X1|X2]) \leftarrow ground(X1), groundArgs(X2)$$

Consider the specialisation of $unify(X, Y, S)$ where X and Y are constrained to be ground. X and Y unify if and only if they are identical.

Can program specialisation automatically produce this result giving a residual program that simply checks identity of the terms? The answer is clearly negative for most available algorithms. With Conjunctive Partial Evaluation or Unfold-Fold transformations it is not so obvious, but we know of no implementations or strategy that would produce the required result automatically. The answer is also negative for the top-down version of the algorithm in this paper (Figure 2). The reason is that the unification of arguments within the terms can depend on the substitutions obtained by unifying other arguments. For instance, unifying $f(X, X)$ and $f(a, b)$ fails even though the arguments unify pairwise, since (assuming the arguments are unified from left to right) the attempted unification of X with b is computed in the context of the substitution $\{X/a\}$ obtained by unifying the first arguments.

The algorithm in Figure 3, which automatically propagates the answer substitutions, returns the following specialised program by specialising with respect to the R-atom $(unify(X, Y, S), U, R)$ where $U = (ground(X), ground(Y), any(S))$ and R is the definition of $ground$ given above.

$$
\begin{array}{l}
unify(struct(X1, X2), struct(X1, X2), []) \leftarrow \\
\quad unifyargs_1(X2, X2, []) \\
unifyargs_1([], [], []) \leftarrow \\
unifyargs_1([struct(X1, X2)|X3], [struct(X1, X2)|X3], []) \leftarrow \\
\quad derefargs_1(X2, X2), \\
\quad derefargs_1(X2, X2), \\
\quad unifyargs_1(X2, X2, []), \\
\quad unifyargs_1(X3, X3, []) \\
derefargs_1([], []) \leftarrow \\
derefargs_1([struct(X1, X2)|X3], [struct(X1, X2)|X3]) \leftarrow \\
\quad derefargs_1(X2, X2), \\
\quad derefargs_1(X3, X3)
\end{array}
$$

This program does no more than check the two terms for identity, and could be further simplified by standard post-processing techniques. In fact, the head of the clause for *unify* already contains identical terms for the two terms to be unified, so that if they are not identical, the call fails immediately. The identity of the terms is propagated through the *msg* operation whereas the groundness of all the sub-arguments is propagated through the regular approximations. The result was achieved in our implementation in 43 msecs on a Sun Ultrasparc workstation.

7.2. Specialisation of a Top-Down Regular Parser

A parser for a small regular language is given by Tamaki and Sato (Tamaki and Sato, 1984a). It handles regular expressions over the language $\{a, b, emp\}$ with the operators $*$ (repetition) and $+$ (union), and brackets. We specialised it for a subset of the language $\{a, b\}$ using only the $*$ operator. The algorithm with answer-propagation successfully produced a parser handling only the reduced language. The need for answer-propagation can be seen by examining the following clause in the parser.

$$term(X, Z) \leftarrow factor(X, Y), term1(Y, Z)$$

Since factors can be of any depth, an approximation of result of parsing any factor in the reduced language is required in Y, in order to specialise $term1(Y, Z)$. The time taken is 17 msecs on a Sun Ultrasparc.

8. Conclusions

The work presented here is based on practical problems arising in the specialisation of semantics-based interpreters (Peralta, 2000), but also illustrates a more general framework for partial evaluation in which abstractions and constraints can be integrated within a specialisation algorithm. Leuschel's framework was used for the top-down version of the algorithm, and we developed a version combining bottom-up (or answer-propagation) as well. The abstract domain (regular approximations in this case) is used to prune resultants and to preserve information during generalisation steps. Arithmetic constraints were already used in a similar role in other systems (Lafave, 1998), (Peralta and Gallagher, 2000).

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.

Staying within the algorithm based on regular approximation, precision can be gained in several ways.

- $\Leftrightarrow$ Nondeterministic RUL programs could be allowed. As seen in Section 2 deterministic RUL programs are less expressive than nondeterministic RUL programs. This might provide an alternative to wrapper functions since non-determinacy allows more distinct "cases" of a structure to be represented.

⇔ Regular approximations have been extended with constraints allowing more precise descriptions of sets of terms, such as sorted lists (Sağlam and Gallagher, 1998). Arithmetic constraints with the convex hull upper bound, and Boolean constraints are potentially useful constraint domains.

⇔ Leuschel's framework (Leuschel, 1998b) allows for the unfolding of R-conjunctions rather than R-atoms. Thus, regular approximation could be combined with Conjunctive Partial Evaluation (Leuschel et al., 1996), potentially allowing greater top-down specialisation than is achieved by our algorithm. We conjecture, however, that Conjunctive Partial Evaluation will yield no more specialisation than the algorithm based on answer-propagation.

8.1. Implementation

The implementation of the algorithm presented here was based on the SP system (Gallagher, 1991) for the basic partial evaluation algorithm and the regular approximation tools (Gallagher and de Waal, 1994) for the operations concerning the handling of RUL programs. The code runs in SICStus Prolog and is available from the authors `http://www.cs.bris.ac.uk/~john/software.html`.

References

Bruynooghe, M.: 1991, 'A Practical Framework for Abstract Interpretation of Logic Programs'. *Journal of Logic Programming* **10(2)**, 91–124.

Codish, M., K. Marriott, and C. Taboch: 2000, 'Improving Program Analyses by Structure Untupling'. *The Journal of Logic Programming* **43**(3), 251–263.

Comon, H., M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: 1999, *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata.

Cousot, P. and R. Cousot: 1992, 'Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation'. In: *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming Leuven,Belgium*, Vol. 631 of *Lecture Notes in Computer Science*. pp. 269–295.

Cousot, P. and R. Cousot: 1995, '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*. La Jolla, California, pp. 170–181.

De Schreye, D., R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen: November 1999, 'Conjunctive Partial Deduction: Foundations, Control, Algorithms, and Experiments'. *Journal of Logic Programming* **41**, 231–277.

de Waal, D. and J. Gallagher: 1992, 'Specialisation of a Unification Algorithm'. In: T. Clement and K.-K. Lau (eds.): *Logic Program Synthesis and Transformation: (LOPSTR-91, Manchester)*.

Gallagher, J.: 1991, 'A System for Specialising Logic programs'. Technical Report TR-91-32, University of Bristol.

Gallagher, J.: 1993, 'Specialisation of Logic Programs: A Tutorial'. In: *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. Copenhagen, pp. 88–98.

Gallagher, J., M. Codish, and E. Shapiro: 1988, 'Specialisation of Prolog and FCP programs Using Abstract Interpretation'. *New Generation Computing* **6**, 159–186.

Gallagher, J. and D. de Waal: 1994, 'Fast and Precise Regular Approximation of Logic Programs'. In: P. Van Hentenryck (ed.): *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*.

Gallagher, J. and L. Lafave: 1996, 'Regular Approximation of Computation Paths in Logic and Functional Languages'. In: O. Danvy, R. Glück, and P. Thiemann (eds.): *Partial Evaluation*, Vol. 1110. pp. 115 – 136.

Gurr, C. A.: 1994, 'Specialising the Ground Representation in the Logic programming Language Gödel'. In: Y. Deville (ed.): *Logic Program Synthesis and Transformation, (LOPSTR'93, Louvain-La-Neuve)*. Springer-Verlag Workshops in Computing.

Heintze, N. and J. Jaffar: 1990, 'A Finite Presentation Theorem for Approximating Logic Programs'. In: *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco*. pp. 197–209.

Janssens, G. and M. Bruynooghe: 1992, 'Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation'. *Journal of Logic Programming* **13**(2-3), 205–258.

Jones, N. D.: 1997, 'Combining Abstract Interpretation and Partial Evaluation'. In: P. Van Hentenryck (ed.): *Symposium on Static Analysis (SAS'97)*. pp. 396–405.

Jones, N. D. and S. S. Muchnick: 1982, 'A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures'. In: A. Press (ed.): *Conference Record of the Ninth Symposium on Principles of Programming Languages*. pp. 66–74.

Kursawe, P.: 1987, 'How to invent a Prolog machine'. *New Generation Computing* **5**, 97–114.

Lafave, L.: October 1998, 'A Constraint-Based Partial Evaluator for Functional Logic Programs and its Application'. Ph.D. thesis, Department of Computer Science.

Leuschel, M.: 1997a, 'Advanced Techniques for Logic Program Specialisation'. Ph.D. thesis, Department of Computer Science.

Leuschel, M.: 1997b, 'Program Specialisation and Abstract Interpretation Reconciled'. Technical Report Report CW 259, Department of Computer Science, K.U. Leuven.

Leuschel, M.: 1998a, 'On the Power of Homeomorphic Embedding for Online Termination'. In: G. Levi (ed.): *Proceedings of the Symposium on Static Analysis (SAS'98)*, Vol. 1503. pp. 230 – 245.

Leuschel, M.: 1998b, 'Program Specialisation and Abstract Interpretation Reconciled'. In: J. Jaffar (ed.): *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*. Manchester, UK, pp. 220–234. Extended version as Technical Report CW 259, K.U. Leuven.

Leuschel, M. and D. De Schreye: September 24-27, 1996, 'Logic Program Specialisation: How To Be More Specific'. In: H. Kuchen and S. Doaitse Swierstra (eds.): *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany*, Vol. 1140 of *Lecture Notes in Computer Science*. pp. 137–151.

Leuschel, M., D. De Schreye, and D. A. de Waal: 1996, 'A Conceptual embedding of folding into partial deduction: towards a maximal integration'. In: M. Maher (ed.): *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*.

Leuschel, M. and B. Martens: 1996, 'Global Control for Partial Deduction through Characteristic Atoms and Global Trees'. In: O. Danvy, R. Glück, and P. Thiemann (eds.): *Partial Evaluation*, Vol. 1110. pp. 263 – 283.

Lloyd, J.: 1987, *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag.

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

Mildner, P.: May 1999, 'Type Domains for Abstract Interpretation: A Critical Study'. Ph.D. thesis, Department of Computer Science, Uppsala University.

Mishra, P.: 1984, 'Towards a theory of types in Prolog'. In: *Proceedings of the IEEE International Symposium on Logic Programming*.

Mogensen, T. Æ..: 1988, 'Partially Static Structures in a Self-Applicable Partial Evaluator'. In: D. Bjørner, A. Ershov, and N. Jones (eds.): *Partial Evaluation and Mixed Computation*. pp. 325–347.

Muthukumar, K. and M. Hermenegildo: 1992, 'Compile-time Derivation of Variable Dependency Using Abstract Interpretation'. *Journal of Logic Programming* **13**(2 and 3), 315–347.

Peralta, J. and J. Gallagher: 2000, 'Imperative Program Specialisation: An Approach Using CLP'. In: A. Bossi (ed.): *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*. pp. 103–118.

Peralta, J., J. Gallagher, and H. Sağlam: 1998, 'Analysis of Imperative Programs through Analysis of Constraint Logic Programs'. In: G. Levi (ed.): *Static Analysis. 5th International Symposium, SAS'98, Pisa*. pp. 246–261.

Peralta, J. C.: 2000, 'Analysis and Specialisation of Imperative Programs: An Approach using CLP'. Ph.D. thesis, University of Bristol, Department of Computer Science (submitted).

Puebla, G., M. Hermenegildo, and J. P. Gallagher: 1999, 'An integration of partial evaluation in a generic abstract interpretation framework'. In: O. Danvy (ed.): *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*. San Antonio, Texas, pp. 75–84.

Reynolds, J. C.: 1969, 'Automatic Construction of Data Set Definitions'. In: J. Morrell (ed.): *Information Processing 68*. pp. 456–461.

Reynolds, J. C.: 1970, 'Transformational Systems and the Algebraic Structure of Atomic Formulas'. In: B. Meltzer and D. Mitchie (eds.): *Machine Intelligence*. pp. 135–151.

Sahlin, D.: 1991, 'An Automatic Partial Evaluator for Full Prolog'. Ph.D. thesis, The Royal Institute of Technology.

Sağlam, H. and J. Gallagher: 1998, 'Constrained Regular Approximation of Logic Programs'. In: N. Fuchs (ed.): *Logic Program Synthesis and Transformation (LOPSTR'97)*.

Sørensen, M. and R. Glück: 1995, 'An algorithm of generalisation in positive supercompilation'. In: J. Lloyd (ed.): *Proceedings of the International Symposium on Logic Programming (ILPS'95)*.

Tamaki, H. and T. Sato: 1984a, 'Enumeration of Success-patterns in Logic Programs'. *Theoretical Computer Science* **34**, 227–240.

Tamaki, H. and T. Sato: 1984b, 'Unfold/Fold Transformation of Logic Programs'. In: S.-A. Tarnlund (ed.): *Proceedings of the Second International Logic Programming Conference*. pp. 127–138.

Tamaki, H. and T. Sato: 1986, 'OLDT Resolution with Tabulation'. In: E. Shapiro (ed.): *Proc. 3rd ICLP, London*, Vol. LNCS 225.

Turchin, V.: 1988, 'The Algorithm of generalization in the supercompiler'. In: D. Bjørner, A. Ershov, and N. Jones (eds.): *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*. pp. 531–549.

Van Hentenryck, P., A. Cortesi, and B. Le Charlier: 1994, 'Type Analysis of Prolog Using Type Graphs'. *Journal of Logic Programming* **22(3)**, 179 – 210.

Vanhoof, W., B. Martens, D. D. Schreye, and K. D. Vlaminck: 1998, 'Specialising The Other Way Around'. In: J. Jaffar (ed.): *Proceedings of the Joint International Conference and Symposium on Logic Programming*. pp. 279–293.

Yardeni, E. and E. Shapiro: 1990, 'A Type System for Logic Programs'. *Journal of Logic Programming* **10(2)**, 125–154.

Appendix

A. Proofs of Correctness Conditions on Operations

The proof of Proposition 13 depends on the establishment of several conditions on the operations appearing in the algorithm of Figure 2. We present these as three Propositions, corresponding to the conditions on **aunf**, **ares** and ω respectively. Note that we consider only definite programs in this paper, and **aunf** is applied only to R-atoms. We assume that the unfolding rule is *fair* (Leuschel, 1998b).

In the proofs, we denote by E_θ a conjunction of equality atoms $x_1 = t_1, \ldots, x_n = t_n$ corresponding to a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$. The goal $\leftarrow E_\theta$ has a refutation with computed answer θ. (The clause $(x = x) \leftarrow$ is assumed to be in each program).

Given a conjunction B, and a program P, let τ be an SLD-tree for $P \cup \{\leftarrow B\}$. Let φ be a substitution (restricted to the variables of B). Clearly, for each SLD-derivation in τ with final goal $\leftarrow G$ and computed answer ρ, we can construct another SLD-derivation of $P \cup \{\leftarrow B, E_\varphi\}$ with final goal $\leftarrow G, E_\varphi\rho$ and computed answer ρ.

We define an SLD-tree τ_φ for $P \cup \{\leftarrow B\varphi\}$, as follows. For each SLD-derivation in τ with final goal $\leftarrow G$ and computed answer ρ, there is an SLD-derivation of $P \cup \{\leftarrow B\varphi\}$ with computed answer σ and final goal $G\sigma$, if and only if $P \cup \{\leftarrow E_\varphi\rho\}$ has a refutation with computed answer σ. Clearly, we could have selected E_φ first in the derivation of

$P \cup \{\leftarrow B, E_\varphi\}$, and the computed answer restricted to the variables of $B\varphi$ would then be σ.

PROPOSITION 18. *Condition on* **aunf**

Let P be a definite program. Then for all R-atoms A and atoms $B' \in \gamma(A)$, there exists an SLD-tree τ for $P \cup \{\leftarrow B'\}$ such that for all substitutions θ and goals $\leftarrow Q$, $B' \Rightarrow_\tau^\theta Q \Leftrightarrow \exists C \in \mathbf{aunf}(A)$ s.t. $B' \Rightarrow_C^\theta Q$.

Proof. Suppose $A = (B, U, R)$ and $B' = B\varphi$ such that $\mathbf{nonempty}(U\varphi)$ (referring to Definition 7). Recall that $\mathbf{aunf}(A) = \{B\rho \leftarrow Q \mid B \Rightarrow_\tau^\rho Q,\ \mathbf{nonempty}_R(U\rho)\}$ (Definition 8).

Proof of $\Rightarrow$:

We are given that $B\varphi \Rightarrow_{\tau_B}^\theta Q$; show that $\exists C \in \mathbf{aunf}(A)$ s.t. $B\varphi \Rightarrow_C^\theta Q$.

By construction of τ_φ, there exists a derivation of $P \cup \{\leftarrow B\}$ in τ with final goal $\leftarrow G$, say, and computed answer ρ say, (where ρ is restricted to variables in B), and $P \cup \{\leftarrow E_\varphi\rho\}$ has a refutation with computed answer θ. Hence $Q = G\theta$. Let $C = B\rho \leftarrow G$ be the resultant of that derivation in τ.

Now, consider the derivation for $\{C\} \cup \{\leftarrow B, E_\varphi\}$; after one derivation step the goal is $\leftarrow G, E_\varphi\rho$. $E_\varphi\rho$ has a refutation with computed answer σ. We could have selected E_φ first, in which case we would have obtained a derivation for $\{C\} \cup \{\leftarrow B\varphi\}$ with computed answer θ and final goal $G\theta = Q$. Hence $B \Rightarrow_C^\theta Q$.

It remains to show that $\mathbf{nonempty}_R(U\rho)$, which is the case since $U\varphi\theta$ is an instance of $U\rho$, and $\mathbf{nonempty}(U\varphi\theta)$ since $\gamma(A)$ is downwards closed. Hence $C \in \mathbf{aunf}(A)$.

Proof of $\Leftarrow$:

Assume that $\exists C \in \mathbf{aunf}(A)$ s.t. $B \Rightarrow_C^\theta Q$ and show that $B\varphi \Rightarrow_{\tau_\varphi}^\theta Q$.

Let $C = B\rho \leftarrow G$. Thus there is a derivation of $\{C\} \cup \{\leftarrow B\varphi\}$ with computed answer θ and final goal $\leftarrow Q$ (where θ is restricted to the variables of $B\varphi$); hence there is a derivation of $\{C\} \cup \{\leftarrow B, E_\varphi\}$ with the final goal $\leftarrow G, E_\varphi\rho$, such that $\leftarrow G, E_\varphi\rho$ has a refutation with computed answer θ (restricted to variables in $B\varphi$. Hence there is a derivation of $P \cup \{\leftarrow B, E_\varphi\}$ with the same final goal and and computed answer substitution.

Then, by construction of τ_φ, there is a derivation of $P \cup \{\leftarrow B\varphi\}$ in τ_φ with final goal Q and computed answer θ. Hence $B \Rightarrow_{\tau_B}^\theta Q$.

PROPOSITION 19. *Condition on* **ares**

Let A be an R-conjunction, $B' \in \gamma(A)$ and C a resultant such that $B \Rightarrow_C^\theta Q$ for some θ. Then $Q \in \gamma(\mathbf{ares}(A, C))$.

Proof. Suppose $A = (B, U, R)$, and $B' = B\varphi$, where $\mathbf{nonempty}(U\varphi)$ and $\forall\sigma.\mathbf{nonempty}(U\varphi\sigma)$.

$B\varphi \Rightarrow_C^\theta Q$, hence there is a derivation of $\{C\} \cup \{\leftarrow B, E_\varphi\}$ with computed answer θ (where θ is restricted to $B\varphi$) and final goal $\leftarrow Q$. Thus there is a derivation of $\{C\} \cup \{\leftarrow B\}$ with final goal G and computed answer ρ, say, such that $Q = G\theta$. Thus $B \Rightarrow_C^\rho G$.

$\mathbf{nonempty}(U\rho)$ holds, since $U\rho\theta$ is an instance of $U\varphi$, and $\forall\sigma.\mathbf{nonempty}(U\varphi\sigma)$. Hence $(G, U', R') \in \mathbf{ares}(A, C)$, where $U' = \mathbf{project}_{vars(G)}(\mathbf{simplify}(U\rho))$.

Finally we verify that $Q \in \gamma((G, U', R'))$ where $Q = G\theta$, which holds if $\mathbf{nonempty}(U'\theta)$. $\mathbf{nonempty}(U'\theta)$ holds if $\mathbf{nonempty}(U\rho\theta)$ holds. The latter holds since $U\rho\theta$ is an instance of $U\varphi$, and $\forall\sigma.\mathbf{nonempty}(U\varphi\sigma)$. Hence $Q \in \gamma((G, U', R'))$.

For the next Proposition, note that in our algorithm, ω has two arguments, and thus we use a modified form of Leuschel's condition.

PROPOSITION 20. *Condition on ω*

Let S, S' be sets of R-conjunctions. Then for all $A \in S$, there exists a sequence $\langle A_1, \ldots, A_n\rangle$ such that for all A_i, $A_i \in \omega(S, S')$, and $\gamma(A) \subseteq \{(Q_1, \ldots, Q_n)\} \mid \forall i : 1 \leq i \leq n, Q_i \in \gamma(A_i)\}$.

Proof. Recall that $\omega(S, S') = S'\nabla\mathbf{partition}(\mathbf{atoms}(S))$. Let $A = (B, U, R) \in S$, where $B = B_1, \ldots, B_n$ and let $\langle A_1, \ldots, A_n\rangle$ be the sequence of R-atoms where $A_i = (B_i, \mathbf{project}_{vars(B_i)}(U), R)$, $1 \leq i \leq n$.

Each atom in the sequence is in $\mathbf{atoms}(A)$. By definition of $\mathbf{partition}$, for each A_j in the sequence, $\mathbf{partition}(\mathbf{atoms}(S))$ contains some R-atom $\hat{A}_j$ such that $\gamma(A_j) \subseteq \gamma(\hat{A}_j)$, since $\mathbf{partition}(S)$ is defined to be a set of R-atoms each of which is an upper bound of some partition of $\mathbf{atoms}(S)$.

Thus, the set $\{(Q_1, \ldots, Q_n)\} \mid \forall i : 1 \leq i \leq n.Q_i \in \gamma(\hat{A}_i)\}$ contains $\gamma(A)$.

Finally, by definition of widening, for each element $\hat{A}$ of $\mathbf{partition}(\mathbf{atoms}(S))$, the set $S'\nabla\mathbf{partition}(\mathbf{atoms}(S))$ contains an R-atom $\hat{A}'$ such that $\gamma(\hat{A}) \subseteq \gamma(\hat{A}')$. This establishes the required result.

B. Code for the Unification Algorithm

```
unify(X,Y,S) :- unify1(X,Y,[],S).

unify(T1,T2,S,S1) :-
        deref(T1,S,B1), deref(T2,S,B2), unify1(B1,B2,S,S1).
```

```
unify1(var(N),var(N),S,S).
unify1(var(N),var(M),S,[var(N)/var(M)|S]) :-
        N \== M.
unify1(var(N),struct(F,Args),S,[var(N)/struct(F,Args)|S]) :-
        not_occurargs(N,Args).
unify1(struct(F,Args),var(N),S,[var(N)/struct(F,Args)|S]) :-
        not_occurargs(N,Args).
unify1(struct(F,Args),struct(F,Args1),S,S1) :-
        unifyargs(Args,Args1,S,S1).

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

deref(var(N),S,B) :-
        binding(var(N),S,B1,V), deref1(V,B1,S,B).
deref(struct(F,Args),S,struct(F,Args1)) :-
        derefargs(Args,S,Args1).

deref1(bound,B1,S,B) :- deref(B1,S,B).
deref1(unbound,B,_,B).


derefargs([],_,[]).
derefargs([X|Xs],S,[B|Bs]) :-
        deref(X,S,B), derefargs(Xs,S,Bs).

binding(var(N),[],var(N), unbound).
binding(var(N),[var(N)/B|_],B, bound).
binding(var(N),[var(M)/_|S],B, V) :-
        M \== N, binding(var(N),S,B,V).

not_occurs(N,var(M)) :- M \== N.
not_occurs(N,struct(_,Args)) :- not_occurargs(N,Args).

not_occurargs(_,[]).
not_occurargs(N,[X|Xs]) :-
        not_occurs(N,X), not_occurargs(N,Xs).
```