Fast and Precise Regular Approximations of Logic Programs

J.P. Gallagher **D.A. de Waal**

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

john@compsci.bristol.ac.uk, andre@compsci.bristol.ac.uk

Abstract

A practical procedure for computing a regular approximation of a logic program is given. Regular approximations are useful in a variety of tasks in debugging, program specialisation and compile-time optimisation. The algorithm shown here incorporates optimisations taken from deductive database fixpoint algorithms and efficient bottom-up abstract interpretation techniques. Frameworks for defining regular approximations have been put forward in the past, but the emphasis has usually been on theoretical aspects. Our results contribute mainly to the development of effective analysis tools that can be applied to large programs. Precision of the approximation can be greatly improved by applying query-answer transformations to a program and a goal, thus capturing some argument dependency information. A novel technique is to use transformations based on computation rules other than left-to-right to improve precision further. We give performance results for our procedure on a range of programs.

1 Introduction

Given a definite logic program P, the problem we address is how to compute a regular approximation of P. An approximation is a program whose least Herbrand model contains the least Herbrand model of P. Regular approximations provide a way of deciding certain properties of a program, since regular programs have a number of decidable properties and associated computable operations [1].

The question of how to define regular safe approximations of a program has been discussed before [25], [35], [18], [13], [20]. Most of this work considered the definition and precision of various different approximations. Little work emphasising efficient, practical implementation has been presented. In this paper we present an algorithm for computing a regular approximation of a logic program, discuss its performance, with possible trade-offs of precision and efficiency. We then show how to increase precision greatly by the use of query-answer transformations, and present results for a range of programs.

Our method of approximation is based on the framework of *bottom-up abstract interpretation* of logic programs [23]. Like [35] and [18], we define an abstraction of the well-known T_P function, say $\mathcal{T}_P$. We define an abstract function $\mathcal{T}_P$ that maps one regular program to another. The least fixed point (*lfp*) of T_P is the least Herbrand model of a program P. By computing $\mathit{lfp}(\mathcal{T}_P)$ we obtain a description of some superset of $\mathit{lfp}(T_P)$.

The important practical questions in such an abstract interpretation are, firstly, whether the least fixed point is computed in a finite number of iterations, and secondly, to compute successive elements of the sequence as efficiently as possible. We use the strongly connected components of the predicate dependency graph to decompose the fixed-point computation into a sequence of smaller computations.

In the next section we define RUL programs, and the abstract semantic function $\mathcal{T}_P$. An important concept here is the *shortening* of an RUL program, which ensures termination of the abstract interpretation. Shortening is an approximation step that introduces recursive definitions. Its properties influence the algorithm greatly, since it is mainly responsible for the rate of convergence of the fixpoint computation. In Section 5 we give the bottom-up algorithm. It is interesting that good results can be achieved using such standard techniques drawn mainly from deductive database methods. In Section 6 the analysis of a program with a goal is discussed. The key idea is the combination of different computation rules (left-to-right and right-to-left) to increase precision. A discussion of related work is in Section 7.

2 Regular Approximations

Definition 2.1 *canonical regular unary clause*
A **canonical regular unary clause** *is a clause of the form*

$$t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n) \qquad n \geq 0$$

where $x_1, \ldots, x_n$ are distinct variables.

Definition 2.2 *canonical RUL program*
A **canonical regular unary logic (RUL) program** *is a finite set of regular unary clauses, in which no two different clause heads have a common instance.*

The class of canonical RUL programs was defined by Yardeni and Shapiro [35]. More expressive classes of regular program can be defined, but this one allows convenient and efficient manipulation of programs. 'RUL program' from now on in this paper will mean 'canonical RUL program'.

Definition 2.3 *regular definition of predicates*
Let $\{p^1/n^1, \ldots, p^m/n^m\}$ be a finite set of predicates. A **regular definition** *of the predicates is a set of clauses $Q \cup R$, where*

- *Q is a set of clauses of form $p^i(x_1, \ldots, x_{n^i}) \leftarrow t_1(x_1), \ldots, t_{n^i}(x_{n^i})$ $(1 \leq i \leq m)$, and R is an RUL program defining $t_1, \ldots, t_{n^i}$; and*
- *no two heads of clauses in $Q \cup R$ have a common instance.*

Definition 2.4 *regular approximation*
Let P be a definite program and P' a regular definition of predicates in P. Then P' is a regular approximation of P if the least Herbrand model of P is contained in the least Herbrand model of P'.

For convenience, abusing notation, an RUL program can be obtained from a regular definition of predicates by replacing each clause $p^i(x_1, \ldots, x_{n^i}) \leftarrow B$ by a clause

$$approx(p^i(x_1, \ldots, x_{n^i})) \leftarrow B$$

where *approx* is a distinguished unary predicate not used elsewhere. Strictly, each predicate p^i/n^i should be replaced by a corresponding function symbol. This transformed program will also be referred to as a regular definition (or approximation), though strictly the original form without the *approx* predicate is meant. This allows us for the remainder of the paper to restrict attention to RUL programs.

3 Operations on RUL Programs

A number of operations and relations on RUL programs are next defined.

Definition 3.1 $success_R(t)$
Let R be an RUL program and let U_R be its Herbrand universe. Let t be a unary predicate. Then $success_R(t) = \{s \in U_R \mid R \cup \{\leftarrow t(s)\} \text{ has a refutation}\}$

Definition 3.2 *inclusion*
Let R be an RUL program, and let t_1 and t_2 be unary predicates. Then we write $t_1 \subseteq t_2$ if $success_R(t_1) \subseteq success_R(t_2)$.

Definition 3.3 *intersection and upper bound of unary predicates*
Let t_1 and t_2 be unary predicates, defined in an RUL program R. Then the **intersection** *$t_1 \cap t_2$ is defined as a predicate t_3, such that $success(t_3) = success_R(t_1) \cap success_R(t_2)$. An* **upper bound** *$t_1 \sqcup t_2$ is defined as a predicate t_3, such that $success_R(t_1) \cup success_R(t_2) \subseteq success(t_3)$.*

Note that the upper bound as defined in general includes the union of t_1 and t_2. Our algorithm computes a tuple-distributive upper bound.

Procedures for checking inclusion, and computing intersection and a suitable upper bound of predicates in RUL programs can easily be defined. The algorithms correspond to well-known algorithms in finite automata theory. If $size(p)$ is number of predicates on which predicate p depends, then the complexity of inclusion, intersection and tuple-distributive upper bound operations on predicates s and t are all $O(size(s) * size(t))$.

Definition 3.4 *upper bound of RUL programs*
Let R and S be RUL programs. Their upper bound $R \sqcup S$ is obtained by the following steps:

1. *For any predicate t that occurs in both R and S (possibly with different definitions), rename t by predicates t^R in R and by t^S in S (where t^R and t^S are new predicates not occurring in R or S. Compute their upper bound $t^R \sqcup t^S$, and call it t.*
2. *Obtain $R \sqcup S$ from $R \cup S$ by replacing the old clauses for t by the newly computed definition (including subsidiary predicates) of t.*

3.1 Shortening of RUL programs

Definition 3.5 *calls, call-chain, depends on*
Let R be a program containing predicates t and s ($t \neq s$). Then t calls s if s occurs in the body of clause whose head contains t. A sequence of predicates $t_1, t_2, \ldots, t_i, \ldots$ where t_i calls t_{i+1} ($i \geq 1$) is a call-chain. We say t depends on s if there is a call-chain from t to s.

Definition 3.6 *$D(t, s)$*
Let R be an RUL program containing predicates t and s ($t \neq s$). Then the relation $D(t, s)$ is true if t depends on s and the set of function symbols appearing in the heads of clauses in the procedure for t is the same as the set of function symbols appearing in the heads of clauses in the procedure for s.

Definition 3.7 *shortening*
Let R be an RUL program, and let t and s be unary predicates defined in R such that $D(t, s)$ holds. Then a program $Sh(R)$ is obtained from R as follows:

- *If $s \subseteq t$ and $t \subseteq s$ then $Sh(R)$ is obtained by replacing all occurrences of s in clause bodies in R by t.*
- *Otherwise, if $s \subseteq t$ then obtain $Sh(R)$ by replacing by t all occurrences of s in clause bodies in R, where that occurrence depends on t and does not depend on s. (I.e. one changes the definition of t, not that of s).*
- *If $s \not\subseteq t$ then compute $r = s \sqcup t$, with definition R_r. Replace all occurrences of t in clause bodies in R by r, giving R'. Then $Sh(R) = R' \cup R_r$. (Thus all calls to t are replaced by calls to r).*

Example 1 *Let $R =$*

$$\{t(a) \leftarrow true,\ t(f(x)) \leftarrow r(x),\ r(a) \leftarrow true,$$
$$r(f(x)) \leftarrow s(x),\ s(b) \leftarrow true\}$$

$D(t, r)$ holds, and $r \not\subseteq t$, so the upper bound $t \sqcup r$ is computed, called q below. So $Sh(R) =$

$$\{t(a) \leftarrow true,\ t(f(x)) \leftarrow q(x),\ q(a) \leftarrow true,\ q(f(x)) \leftarrow q_1(x),\ q_1(a) \leftarrow true,$$
$$q_1(b) \leftarrow true,\ q_1(f(x)) \leftarrow s(x),\ s(b) \leftarrow true,\ r(a) \leftarrow true,\ r(f(x)) \leftarrow s(x)\}$$

Definition 3.8 *shortened RUL program*
A **shortened** *program R is one in which there are no two predicates t and s such that $D(t, s)$ holds.*

A shortened program can be obtained from R by performing a finite number of applications of Sh to R, i.e. computing $Sh^n(R)$ for some finite n. This is proved in [16].

Definition 3.9 *Let R be an RUL program. Then* $\mathbf{short}(R) = Sh^n(R)$ *such that $Sh^n(R)$ is shortened.*

The main justification for shortening is to limit the number of RUL programs (see Lemma 3.2), and thus ensure termination of our approximation procedure. We have seldom found a serious lack of expressiveness imposed by shortening, though nested occurrences of the same function symbol are (sometimes) merged. In terms of finite automata, a shortened RUL program corresponds to an automaton in which no two connected states have the same set of labels on transitions leading from them. Shortening also puts RUL definitions into a minimal form and this leads to more efficient processing.

Shortening can introduce recursive definitions. For example, the set of atoms

$$\{p(a), p(f(a)), p(f(f(a)))\}$$

can be described by the RUL program

$$\{p(a) \leftarrow true, p(f(x)) \leftarrow t_1(x), t_1(a) \leftarrow true, t_1(f(x)) \leftarrow t_2(x), t_2(a) \leftarrow true\}$$

In this program, $D(p, t_1)$ holds, and $t_1 \subseteq p$. Its shortened form is thus

$$\{p(a) \leftarrow true, p(f(x)) \leftarrow p(x), t_1(a) \leftarrow true, t_1(f(x)) \leftarrow t_2(x), t_2(a) \leftarrow true\}$$

Other kinds of shortening are possible. In [5] an operation on "type graphs" is used, that has some relation to our operation. Other possible shortenings are discussed in Section 7. Study of suitable shortenings remains a key issue, since it is here that tradeoffs of precision and complexity are focussed.

The next lemma (proved in [16]) states that shortening increases the size of the success set of any predicate that occurs in both R and $Sh(R)$.

Lemma 3.1 *Let R be an RUL program containing a predicate t such that t occurs both in R and $Sh(R)$. Then $success_R(t) \subseteq success_{Sh(R)}(t)$.*

Let U_F be some Herbrand universe, constructed from a finite set of constants and function symbols $F = \{f_1/n_1, \ldots, f_m/n_m\}$. Let R be any program whose Herbrand universe is a subset of U_F. Then clearly $success_R(t) \subseteq U_F$, for each unary predicate t in R. A special predicate called *any* is such that $success_R(any) = U_F$.

The next lemma shows that where R is a shortened RUL program with some fixed Herbrand universe, there is only a finite set of possible sets $success_R(t)$. This property is later used to show the termination of the algorithm for deriving a regular approximation of a program.

Lemma 3.2 *Let $F = \{f_1/n_1, \ldots, f_m/n_m\}$ be a finite set of constant and function symbols. Then there is a finite number of different sets of the form $success_R(t)$, where R is any RUL program containing constants and functions from F (and no others), and t is a unary predicate in R.*

Proof. (sketch)
A deterministic finite automaton corresponding to a program can be constructed. It can be seen that only a finite number of distinct automata can be constructed from RUL programs with an upper bound on the length of acyclic call-chains. The maximum length of acyclic call-chains in a shortened RUL program with m function symbols of non-zero arity is $2^m - 1$. The result follows. $\square$

4 Computing a Regular Approximation of a Program

In this section the abstract semantic function $\mathcal{T}_P$ is defined. First the *regular solution of a definite clause* is defined, by means of the following operator **solve**. For reasons of space we give only a sketch. The full definition is in [15]. A related procedure was used by Frühwirth [12].

The procedure **solve**(C, R) takes a clause C and a regular definition of predicates, R. The procedure returns a regular definition of the predicate in the head of C. The body of C is "solved" in R. This involves unfolding the body until it consists of unary atoms with variable arguments, followed by intersection of predicates with the same argument. The terms in the head of the clause are then given definitions in terms of the unary body predicates.

If the procedure returns $\emptyset$ this is interpreted as failure. This can happen either because unfolding fails, or because some intersection is empty. If it does not fail, the procedure yields a unique result which is an RUL program.

Example 2 *Let the clause C be*

$$append([x|xs], ys, [x|zs]) \leftarrow append(xs, ys, zs)$$

and let $R = \{approx(append(x, y, z)) \leftarrow t_1(x), any(y), any(z), \quad t_1([]) \leftarrow true\}$.
solve(C, R) *yields the following program.*

$$\begin{array}{l}\{approx(append(x, y, z)) \leftarrow t_2(x), any(y), t_3(z),\\ t_1([]) \leftarrow true,\\ t_2([x|xs]) \leftarrow any(x), t_1(xs),\\ t_3([z|zs]) \leftarrow any(z), any(zs)\}\end{array}$$

Definition 4.1 *the function $\mathcal{T}_P$*
Let P be a definite program. Let D be an RUL program (a regular definition of predicates in P). A function $\mathcal{T}_P(D)$ is defined as follows:

$$\mathcal{T}_P(D) = \mathbf{short}(\bigsqcup\{\ \mathbf{solve}(C, D) \mid C \in P\ \}\bigsqcup D)$$

That is, **solve** *is applied to each clause, their upperbound with D is computed and the whole result shortened.*

Definition 4.2 $\gamma(R)$
Let R be an RUL program. The set $success_R(approx)$ is called the concretisation of R, and is referred to as $\gamma(R)$.

Let P be a program with Herbrand base B_P and let R be a regular definition of predicates in P. Then $\gamma(R)$ can be identified with some subset of B_P (allowing for the shortcut in notation mentioned earlier that confuses predicate and function symbols). An RUL program (containing a predicate *approx*) can be thought of as describing some Herbrand interpretation.

Our next problem is to establish that $\mathcal{T}_P$ has the required properties, namely,

- that its least fixed point exists and is computable, and
- that it is a safe abstraction of T_P.

Space does not allow the details, which are based on well-known methods from abstract interpretation (e.g. [9], [4], [24]). The following is the required safety result.

Lemma 4.1 *Let P be a definite program. Then $\forall n(T_P^n(\emptyset) \subseteq \gamma(\mathcal{T}_P^n(\emptyset)))$.*

The fixed point of $\mathcal{T}_P$ is found in a finite number of iterations, because of Lemma 3.2. From this it follows that $lfp(T_P) \subseteq \gamma(lfp(\mathcal{T}_P))$.

Prolog built-ins are handled either by defining some basic regular types (such as integer expressions) or simply by using the trivial approximation *any* for each built-in argument.

5 Efficient Implementation

5.1 Naive and Semi-Naive Evaluation

In the field of deductive databases, considerable investigation has been made of the efficient evaluation of least fixed points for recursive query answering [34]. Also, in abstract interpretation, generic fixed-point-finding algorithms have been defined which use related techniques [22].

We cannot give a detailed description here, but simply point to the main techniques we have used. Some similar optimisation methods were used in another fast bottom-up abstract interpretation [8]. The key is to use the ideas of *semi-naive* evaluation [34] to exploit the new information Δ_i derived on the ith iteration, to limit the work in the $i + 1$th iteration. The exploitation of the Δ_i part can be effected in various ways.

In our algorithm we do not use any dynamic dependency information (as is used in other abstract interpretation frameworks [22], [27] and [4]). We apply the idea of semi-naive evaluation in a simple but effective way. When computing $\mathcal{T}_P(D)$ on some iteration, we identify the predicates whose approximation are changed by that application. On the subsequent iteration, we only look at clauses whose bodies contain one of the changed predicates. All other clauses can be ignored since they will obviously return exactly the same as on the previous iteration.

In effect, the definition of $\mathcal{T}_P$ is modified to include a set of "changed predicates" S as an argument, and to return another set of changed predicates in its result. Let C_S stand for any clause whose body contains a predicate from the set S. The definition is then as follows:

$$\begin{array}{rl} \mathcal{T}_P(D,S) = & (D_1, S_1) \ \text{ where} \\ & D_1 = \mathbf{short}(\bigsqcup \left\{ \ \mathbf{solve}(C_S, D) \ \middle| \ C_S \in P \ \right\} \bigsqcup D) \\ & S_1 = \left\{ \ p \ \middle| \ \text{ definition of } p \text{ changed from } D_1 \text{ to } D_2 \ \right\} \end{array}$$

A further optimisation in our algorithm is to apply **short** only to the predicates in S_1. These are major optimisations for most programs. Furthermore, a sufficient condition for termination is that S_1 is empty on some iteration.

The question of how to access efficiently all clauses whose bodies contain a given predicate can be solved in different ways, perhaps using an index from predicates to clause bodies. However, using the optimisation discussed in the next section based on strongly connected components, the number of clauses that need to be examined on each iteration is cut down, so indexing is not a great advantage.

5.2 Use of Strongly Connected Components

An optimisation that has appeared in the literature in various settings, but seems not so well-known as the semi-naive algorithm, is the use of strongly connected components (SCCs) [26], [31]. Given the predicate-dependency graph of a program, an SCC is a set of predicates all of which depend on each other. A linear algorithm to decompose a graph into its SCCs was discovered by R. Tarjan [33]. The set of SCCs of a graph themselves form an acyclic graph, and this set can then be topologically sorted to give a sequence of SCCs.

The fixed-point-finding algorithm can then be decomposed into a sequence of fixpoint computations, starting from the bottom of the SCC graph and working upwards. In each SCC, only the clauses for the predicates in that SCC are considered, and a fixed point for the meaning of the SCC predicates is computed. By working through the sorted sequence of SCCs, it is assured that predicates lower in the dependency graph are computed before the predicates that depend on them.

In typical user programs, the number of mutually recursive predicates in an SCC is rarely more than two. So this means that relatively few clauses need to be examined on each iteration of the algorithm, with consequent speed-up. (In programs that are generated automatically, such as the query-answer transforms discussed in Section 6 we have found that SCCs are often much larger).

The SCCs can also be used to identify which predicates are recursive and which are not. When computing the meaning of a non-recursive predicate (whose SCC is obviously a singleton) no fixpoint computation and no shortening are needed.

Note that the SCC optimisation involves some trade-off. Firstly, the SCCs have to be computed (a linear computation, as observed already). Secondly, although each iteration is simpler, there may be more iterations in total over the whole algorithm, since some SCCs are independent and their results would be computed simultaneously without use of SCCs. This suggests experimenting with merging SCCs that are independent.

Other dependency graphs could be used apart from the predicate dependencies. For example, a clause dependency graph can be defined, and its SCCs exploited in a similar way. However, although this gives finer grain dependency information, the optimisations obtainable here would appear to be similar to those obtained by the semi-naive algorithm.

5.3 The Effect of Shortening

Shortening determines the number of iterations that are required to reach a fixed point, since recursive approximations are introduced only by shortening. One could gain precision at the expense of complexity and vice versa, by employing different shortenings. One could force recursive approximations to be generated from weaker conditions on the RUL clauses and hence accelerate termination of the approximation. This is further discussed in Section 7. Secondly, shortening has the effect of minimising the size of the RUL definitions, and so the cost of basic operations such as intersection, which depends on the size of definitions, is minimised.

5.4 Factors Influencing Performance

Complexity analysis of this method can establish worst cases. Shortening places a worst-case upper bound on the size of RUL definitions. The depth of call-chains in a language with m function symbols of non-zero arity is at most $2^m - 1$, as noted in Lemma 3.2. This implies exponential upper-bounds on the cost of operations such as intersection, and on the number of iterations required to reach a fixed point.

Average complexity is influenced by a number of factors. In practice average cases are quite tractable, since the number of function symbols of non-zero arity that can appear in a given argument position of a predicate is the relevant value of m, and this tends to be small.

Our experiments indicate that the performance of the algorithm is affected roughly linearly by the number of SCCs, the average size of clauses and the average size of SCCs. The average number of times that predicates within an SCC occur in each other's clauses would appear to be another important factor, since the effectiveness of our semi-naive algorithm depends on this.

6 Experimental Results

In this section we demonstrate how the accuracy of the regular approximations can be improved by using of a class of program transformations that includes the "magic set" and "Alexander" transformations. We call these "query-answer" transformations. These transformations were introduced as recursive query evaluation techniques [2], [30], but have since been adapted for use in program analysis [6], [11], [21], [32], [16]. They allow top-down computation to be performed in a bottom-up manner.

Bottom-up analysis does not take into account possible queries to a program. By analysing the computation of a general call to a predicate $p(x_1, \ldots, x_n)$ we may get a more precise result for that predicate than by analysing the program bottom-up. This was pointed out in [16] and discussed in detail in [7]. Examples of query-answer transformations can be found in in [16] and [14].

Consider the following permutation program that computes permutations of lists of integers. ($integer(X)$ is a built-in predicate for which a predefined approximation $numeric(X)$ is derived).

$perm(X, Y) : -intlist(X), permutation(X, Y).$
$permutation([], []).$
$permutation(X, [U|V]) : -delete(U, X, Z), permutation(Z, V).$
$delete(X, [X|Y], Y).$
$delete(X, [Y|Z], [Y|W]) : -delete(X, Z, W).$
$intlist([]).$
$intlist([X|Xs]) : -integer(X), intlist(Xs).$

The following program is the regular approximation produced by our analysis (showing the result for *perm* and *permutation* only). The t-predicates, *any* and *numeric* are generated by the approximation procedure.

$perm(A1, A2) : -t42(A1), t28(A2).$
$permutation(A1, A2) : -t21(A1), t28(A2).$
$t28([]) : -true.$
$t28([A1|A2]) : -any(A1), t28(A2).$
$t42([]) : -true.$
$t42([A1|A2]) : -numeric(A1), t42(A2).$
$t21([]) : -true.$
$t21([A1|A2]) : -any(A1), any(A2).$

Although the second argument of *perm* is shown to be a list, it cannot be inferred to be a list of integers. The results for the first argument of *permutation* are very imprecise. Note that no other method of regular approximation in the literature based on bottom-up approximation, with no argument dependency information, would give better results than this.

6.1 Increasing Precision with Query-Answer Transformation

Assuming a left-to-right computation rule, a "magic" style transformation of the program with respect to the goal $\leftarrow perm(x, y)$ is generated and analysed. In the transformed program, the predicate *perm_ans* gives the answers for *perm*. A regular approximation of the "magic" transformed program, showing this predicate is:

$perm\_ans(A1, A2) :- t23(A1), t23(A2).$
$t23([]) :- true.$
$t23([A1|A2]) :- numeric(A1), t23(A2).$

It is now much more precise than the original result. The second argument of *perm* is shown to be a list of numeric values. Effectively, variable sharing information in the original program has been made explicit, detecting that the *perm* predicate returns a list of integers when given a list of integers.

For some programs the regular approximation procedure might still not be unable to infer optimal information even with a "magic" transformed version of a program. An example of such a program is a similar *perm* program, but with the second argument known to be a list of integers but not the first argument. In this case the transformation above would not detect that the first argument must also be a list of integers. We now describe an improvement over the "magic set" transformation used so far to enable us to infer even more detailed information.

6.2 Further Precision by Adding Right-to-Left Computation

As we are only interested in the success set of a program, not its computation tree, the left-to-right computation rule we assumed is not a prerequisite. An analysis assuming a right-to-left computation rule (or any other computation rule) is just as valid. The improved ''magic" transformation generates two "answer" and "query" predicates for each predicate, one for the left-right rule and one for the right-left. The final answers are obtained by intersecting the left and right answers. For each predicate a clause is added to the query-answer transformed program to compute this. For example for the *perm* predicate the following clause is added

$$perm\_ans(X, Y) :- perm\_left\_ans(X, Y), perm\_right\_ans(X, Y).$$

The intersection of regular approximations using different computation rules might give even more precise regular approximations, as information from the "right" of a clause may now be propagated to the literals to its left. The use of additional right-to-left analysis has some relation to "reexecution" to gain precision in other abstract interpretation frameworks.

Analysing computations using two computation rules might seem like an unacceptable overhead. However near-optimal or optimal information (within the limits of RUL programs) can often be computed by this means. Furthermore the two approximations using the two computation rules can be done independently (this is detected by the decomposition into SCCs, Section 5) and are intersected only at the final step. The approximation time will therefore approximately double, but it might be an acceptable price to pay for the increase in precision.

A regular approximation of the permutation program with second argument known to be a list of integers using the improved magic set transformation enables us to infer the most precise RUL approximation possible for this program. Different intermediate levels of precision are also possible by using simplified versions of the improved "magic" transformation presented here with a corresponding decrease in execution times.

6.3 Benchmarks

We now give some performance results for some typical examples used in the abstract interpretation literature. The query-answer transformations were performed with respect to a "top" goal for each program. A Sun Sparc-10 running Sicstus Prolog was used in all the experiments and all timings are in seconds. The number of clauses in brackets is the number of clauses in the left-transformed program. (The number of query clauses generated is equal to the number of body literals in the original program). The number of predicates in the left and left+right programs are respectively 2 and 5 times the number of predicates in the original. The number of clauses in the left+right program is $2k + p$ where k is the number of clauses in the left-transformed version and p is the number of predicates in the original program.

Benchmarks					
	Number of	Number of	Bottom-		Left+
Program	Clauses	Predicates	Up	Left	Right
Board Cutting	110 (238)	37	1.08	7.22	13.88
Scheduling	64 (137)	34	0.61	4.66	8.56
Gabriel Benchmark	46 (84)	20	0.33	1.22	3.30
Peephole Optimiser	228 (319)	22	3.63	7.63	19.88
Press	156 (282)	50	1.33	8.34	19.65
Quicksort	8 (15)	4	0.06	0.13	0.43
N-Queens	10 (19)	5	0.07	0.14	0.52
Prolog Tokeniser	169 (358)	43	1.04	7.99	21.44

Table 1: Approximation Times for Benchmark Programs

The increase in complexity due to the "query-answer" transformation is significant but not prohibitive, and is 3 to 7 times more expensive than the bottom-up analysis for these benchmarks. This represents the overhead for top-down over bottom-up. The increase for additional right-to-left analysis is roughly a further factor of 2 to 3. We have not yet made detailed analysis of the precision gains from the left+right transformation.

7 Discussion

7.1 Other Approaches to Regular Approximation

Mishra [25] suggested using regular structures to approximate the success set of a program, and called these approximations "types". The approximation was obtained by solving a set of recursive equations derived from the program. This general approach was continued by Heintze and Jaffar [18] and by Frühwirth *et al.* [13]. Both of these papers shed much light on the general problem of defining regular approximations, and the precision of different classes of approximation.

The approach to regular approximation based on abstract interpretation of the T_P function was initiated by Yardeni and Shapiro [35], but theirs was a theoretical construct that did not yield an explicit form. Jones [20] also gave an algorithm to compute a regular tree-grammar which approximated a set of rewrite rules. This is a fixpoint algorithm with a strong connection to abstract interpretation. Frühwirth *et al.* showed the relation (for logic programs) between the approaches based on set equations and on abstract interpretation.

The approaches based on set equations seem at first sight to be more direct, but in practice the difficult part is putting the equations into some usable solved form. Heintze [17] outlines an implementation of such an algorithm, and gives experimental evidence that it is efficient for fairly small programs. He also employs a construction for simulating the top-down left-to-right computation rule, which appears to be identical to well-known "magic set" methods used in this paper. Algorithms for getting solved form have similarities to the fixpoint computations of abstract interpretation methods. There may be deeper connections between these streams yet to be investigated. Very recently, some other work in the abstract interpretation framework shows that fast and precise results can be obtained [19]. This work uses a generic top-down abstract interpretation framework, and is a useful contrast to our approach using the bottom-up framework with query-answer transformations. Their approach uses a "widening" operator to ensure termination.

The phrase "type inference" was used (in a non-standard way) by several authors when speaking of regular approximations. Zobel [36], Frühwirth [12] and Reddy [29] defined algorithms that infer descriptions of success "types". Comparison of their results with ours is difficult, but for small examples is similar.

Domains for abstract interpretation of logic programs have been defined that include type-like structures by Kanamori [21] and Bruynooghe and Janssens [5]. The former is not comparable to ours, since it relies on a given set of basic types whose properties are built in to the abstract interpretation. The latter also uses given basic types in the domain, as well as sharing information, but it seems that one could remove these from their abstract domains and extract regular types similar to ours.

7.2 Precision

In bottom-up regular approximation the main loss in precision is due to loss of information about dependencies between arguments. Other aspects of precision, such as those discussed by [18] and [13] are comparatively insignificant. In view of this it is worthwhile to sacrifice a little precision (due to our shortening operation) in return for speed. In particular, derivation of non-deterministic regular programs (in which one dropped the condition that clause heads have no common instances) would greatly increase the cost of basic operations on RUL programs.

A far more important aspect of precision is to capture argument dependencies. Our approach to this is to analyse with respect to a given goal, by performing a query-answer transformation. We have shown that argument dependencies can be captured, and precision is thus greatly increased. Optimal results are often obtained. The alternative is to develop richer abstract domains incorporating sharing information (as in [5]) and it is not clear yet whether one approach is better than the other.

Other shortenings on RUL programs are possible that could increase precision. We have experimented with a shortening that introduces recursive approximations directly from the recursive structure of the program being analysed (this was suggested by the method in [13]). Another possibility is to look ahead further than the set of functions symbols occurring in clause heads when shortening. Similar ideas occur in machine learning, and in fact our shortening was originally inspired by a procedure for inferring automata in [3]. An operation on type graphs is defined in [5]. Repetitions of function symbols on a branch of a type graph are merged into a loop. Our shortening is similar in spirit but is usually more precise.

7.3 Applications

The main purpose of this paper is not to describe applications of regular approximations, of which there are many. Type inference and related debugging applications are discussed in [25], [35], and elsewhere. These applications are well presented by Naish [28]. Regular approximation can be combined with the declaration of intended types (stated as regular programs) and program errors detected. Compile-time optimisation using information about argument structure is another important application.

The combination of regular approximation with partial evaluation was developed by us in [14]. Deletion of *useless clauses*, detected by a regular approximation, can greatly improve the results of partial evaluation. This was applied with good results in specialising theorem provers [10].

Acknowledgements

This work is supported by ESPRIT Project PRINCE (5246). We thank the ICLP referees for their suggestions and corrections.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Adddison-Wesley, 1974.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5^{th} ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

[3] A. Biermann. Fundamental mechanisms in machine learning and inductive inference. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence: An Advanced Course*, Springer-Verlag, 1987.

[4] M. Bruynooghe. A practical framework for abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

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

[6] M. Codish. *Abstract Interpretation of Sequential and Concurrent Logic Programs.* PhD thesis, The Weizmann Institute of Science, 1991.

[7] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. *Top-Down vs Bottom-Up Analysis of Logic Programs – Closing the Circle.* Technical Report Report CW 177, Department of Computer Science, K.U. Leuven, 1993.

[8] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*, MIT Press, 1993.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[10] D.A. de Waal and J. Gallagher. Logic program specialisation with deletion of useless clause (poster abstract). In D. Miller, editor, *Proceedings of the 1993 Logic programming Symposium, Vancouver*, page 632, MIT Press, (full version appears as CSTR-92-33, Dept. Computer Science, University of Bristol), 1993.

[11] S. Debray and R. Ramakrishnan. *Canonical computations of logic programs.* Technical Report, University of Arizona-Tucson, July 1990.

[12] T. Frühwirth. Type inference by program transformation and partial evaluation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, MIT Press, 1989.

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

[14] J. Gallagher and D.A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 151–167, Springer-Verlag, 1993.

[15] J. Gallagher and D.A. de Waal. *Fast and Precise Regular Approximation of Logic Programs.* Technical Report TR-93-19, Dept. of Computer Science, University of Bristol, 1993.

[16] J. Gallagher and D.A. de Waal. *Regular Approximations of Logic Programs and Their Uses.* Technical Report CSTR-92-06, University of Bristol, March 1992.

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

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

[19] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. *Type Analysis of Prolog Using Type Graphs.* Technical Report, Brown University, Department of Computer Science, December 1993.

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

[21] T. Kanamori. *Abstract interpretation based on Alexander templates.* Technical Report TR-549, ICOT, March 1990.

[22] B. le Charlier, K. Musumbu, and P. van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 64–78, MIT Press, 1991.

[23] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming, Washington*, August 1988.

[24] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational abstract interpretation logic programs. *ACM Transactions on Programming Languages and Systems*, to appear, (also appears as Technical Report 92/20, Dept. of Computer Science, University of Melbourne).

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

[26] K. Morris, J.D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In E. Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, pages 554–568, Springer-verlag Lecture Notes in Computer Science, 1986.

[27] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189, MIT Press, October 1989.

[28] L. Naish. *Types and the intended meanings of logic programs.* Technical Report, University of Melbourne, Department of Computer Science, 1990.

[29] U. Reddy. Notions of polymorphism for predicate logic programming. In R.A. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, 1988.

[30] J. Rohmer, R. Lescœr, and J.-M. Kerisit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4, 1986.

[31] D. Sahlin and T. Sjoland. Towards an Analysis for the AKL Language. In *1993 ICLP Workshop on Concurrent Constraint Programming*, SICS, June 1993.

[32] H. Seki. On the power of Alexander templates. In *Proceedings of the 8^{th} ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, Pennsylvania*, 1989.

[33] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[34] J.D. Ullman. *Principles of Knowledge and Database Systems; Volume 1.* Computer Science Press, 1988.

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

[36] J. Zobel. Derivation of polymorphic types for Prolog programs. In R.A. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, MIT Press, 1988.