Regular Approximations of Logic Programs and Their Uses[1]

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

March 1992

CSTR-92-06

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

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

[1]Work supported by ESPRIT Project PRINCE (5246)

Abstract

Regular approximations of logic programs have a variety of uses, including static analysis for debugging, program specialisation, and machine learning. An algorithm for computing a regular approximation of a normal program is given, and some applications are discussed. The analysis of a "magic set" style of transformation of a program P can be used to derive more precise approximations than can be obtained from P itself. The approximation algorithm given here can also be applied to Prolog programs.

1 Regular Logic Programs

Regular structures can be used to approximate programs as shown, for example, in [11], [15], [18]. Regular sets and regular languages have a number of decidable properties and associated computable operations [1], so that approximations of program meanings expressed as regular structures can conveniently be analysed and manipulated.

In this paper an algorithm is given, which takes a normal logic program and returns a regular program. The regular program is a safe approximation of the original program, in a sense to be defined shortly. Several uses of regular approximations are then discussed.

The class of Regular Unary Logic programs was defined by Yardeni and Shapiro [18]. A slightly more restricted class is defined below; however it too will be referred to as the class of Regular Unary Logic programs.

Definition 1.1 *regular unary clause*

A **regular unary clause** *is of the form* $t_0(f(x_1,\ldots,x_n)) \leftarrow t_1(x_1),\ldots,t_n(x_n)$*, where* $x_1,\ldots,x_n$ *are distinct variables.*

Definition 1.2 *regular unary logic program (RUL program)*

A **regular unary logic program** *is a finite set of regular unary clauses, in which there are no two different clauses having the same predicate and function symbols in the heads.*

In the definition in [18], the restriction on RUL programs is that the clause heads are mutually non-unifiable, and every variable in a clause head appears exactly once in the clause body. Thus for example the clauses $\{p(f(a)) \leftarrow true, p(f(b)) \leftarrow true\}$ would be permitted by [18], but not by Definition 1.2, since $p/1$ and $f/1$ occur in both clause heads. The more restricted definition allows somewhat more convenient manipulation of RUL programs.

2 Approximation of Logic Programs

The aim of static analysis of a program is to derive a safe approximation of the meaning or behaviour of the program. The field of abstract interpretation provides a framework defining safe approximation and abstraction of program semantics, into which the present work could be fitted. But for the present purposes, the notion of safe approximation is defined directly with respect to the procedural semantics of logic programs, and much of the more general framework of abstract interpretation is omitted.

The usual definitions of definite and normal programs and goals, and of SLD and SLDNF derivations are used [13].

Definition 2.1 *safe approximation*

Let P,P' *be normal programs. Then* P' *is a* **safe approximation** *of* P *if for all ground atoms* A, $P \cup \{\leftarrow A\}$ *has an SLDNF refutation implies* $P' \cup \{\leftarrow A\}$ *has an SLDNF refutation.*

This definition states that the success set of a logic program is being approximated, that is, P' has a greater success set than P. Therefore, any property that holds for of all elements of the success set of P' holds also for all elements of the success set of P.

Definition 2.2 *regular definition of predicates*

Let P be a normal program containing predicates $\{p^1/n^1, \ldots, p^m/n^m\}$. A **regular definition** *of the predicates in P is a set of clauses $R \cup Q$, where Q is a set of clauses of form $p^i(x_1, \ldots, x_{n^i}) \leftarrow t_1(x_1), \ldots, t_{n^i}(x_{n^i})$, and R is an RUL program defining $t_1, \ldots, t_{n^i}$. If $R \cup Q$ is a safe approximation of P it is called a* **regular approximation** *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 properties of and operations on RUL programs are next defined.

Definition 3.1 *$success_R(t)$*

Let R be a set of regular unary clauses containing a predicate t. The set of terms $success_R(t) = \{s \mid s \text{ a ground term}, R \cup \{\leftarrow t(s)\} \text{ succeeds}\}$

Definition 3.2 *inclusion*

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

The property $t_1 \subseteq t_2$ is true if:

> *for every clause $t_1(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n)$ in R there is a clause $t_2(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n)$ in R, and $s_i \subseteq r_i$, $1 \leq i \leq n$.*

This can be checked finitely.

Definition 3.3 *intersection of unary predicates*

Let R be an RUL program, and let t_1 and t_2 be unary predicates defined in R. Then the **intersection** *$t_1 \cap t_2$ is defined as a predicate t_3 with definition R', such that $success_{R \cup R'}(t_3) = success_R(t_1) \cap success_R(t_2)$.*

A definition for the intersection of two unary predicates can be computed as follows. Given t_1 and t_2, the intersection is:

- *t_1 if $t_1 \subseteq t_2$, or*
- *t_2 if $t_2 \subseteq t_1$, or*

- *a predicate t_3 otherwise, where t_3 is defined as follows:*
  - *If $t_1(f(x_1,\ldots,x_n)) \leftarrow s_1(x_1),\ldots,s_n(x_n)$ is in R and $t_2(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n)$ is in R then there is a clause $t_3(f(x_1,\ldots,x_n)) \leftarrow q_1(x_1),\ldots,q_n(x_n)$, where q_i is defined as the intersection of s_i and r_i.*

Definition 3.4 *upper bound*

Let R be an RUL program, and let t_1 and t_2 be unary predicates defined in R. Then an **upper bound** *$t_1 \sqcup t_2$ is defined as a predicate t_3 with definition R', where $success_R(t_1) \cup success_R(t_2) \subseteq success_{R\cup R'}(t_3)$.*

A definition for an upper bound of two unary predicates can be computed as follows. Given t_1 and t_2, an upper bound is:

- *t_2 if $t_1 \subseteq t_2$, or*
- *t_1 if $t_2 \subseteq t_1$, or*
- *a predicate t_3 otherwise, where t_3 is defined as follows:*
  - *If $t_1(f(x_1,\ldots,x_n)) \leftarrow s_1(x_1),\ldots,s_n(x_n)$ is in R and $t_2(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n)$ is in R then there is a clause $t_3(f(x_1,\ldots,x_n)) \leftarrow q_1(x_1),\ldots,q_n(x_n)$, where $q_i = s_i \sqcup r_i$;*
  - *If $t_1(f(x_1,\ldots,x_n)) \leftarrow s_1(x_1),\ldots,s_n(x_n)$ is in R and there is no clause $t_2(f(x_1,\ldots,x_n)) \leftarrow B$ in R, then there is a clause $t_3(f(x_1,\ldots,x_n)) \leftarrow s_1(x_1),\ldots,s_n(x_n)$.*
  - *If $t_2(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n)$ is in R and there is no clause $t_1(f(x_1,\ldots,x_n)) \leftarrow B$ in R, then there is a clause $t_3(f(x_1,\ldots,x_n)) \leftarrow r_1(x_1),\ldots,r_n(x_n)$.*

Note that t_3 as computed above does not necessarily give the union of the results of t_1 and t_2, but in general includes the union. Although unions of regular sets are regular and can be computed, they cannot be expressed always as RUL programs. (They could be if we used the less restricted definition of RUL programs given in [18]).

Next, an operation on regular predicate definitions is defined, which is designed to derive programs of a certain standard form. The idea is to limit the number of predicates depending on a given predicate.

Definition 3.5 *$D(t,s)$*

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 a subset of the set of function symbols appearing in the heads of clauses in the procedure for s.

Definition 3.6 *normalisation*

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 $N(R)$ is obtained from R as follows:

- *If $s \subseteq t$ and $t \subseteq s$ then replace by t all occurrences of s in clause bodies in R, and delete the procedure for s from R, yielding $N(R)$.*
- *Otherwise, if $s \subseteq t$ then replace by t all occurrences of s in clause bodies in R that depend on t and do not depend on s. If after the replacement no occurrences of s are present, then delete the clauses for s from R, yielding $N(R)$.*
- *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, and delete the procedure for t, yielding R'. Then $N(R) = R' \cup R_r$.*

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 of t and r is computed. This is the predicate q defined as:

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

So $N(R) =$

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

Definition 3.7 *normalised RUL program*
A **normalised** *program R is one in which there are no two predicates t and s such that $D(t, s)$ holds.*

A normalised program can be obtained from R by performing a finite number of applications of N to R, i.e. computing $N^n(R)$ for some finite n. This is shown by the following lemma. First, a chain of dependent predicates is defined.

Definition 3.8 *Let R be an RUL program. Then a sequence of distinct predicates $t_1, t_2, \ldots, t_n$ is called a* **maximal dependency path** *if t_i depends on t_{i+1}, $(1 \leq i < n)$, and t_n depends only on zero or more predicates in $t_1, t_2, \ldots, t_n$.*

Lemma 3.9 *Let R be an RUL program. Then for some $n \geq 0$, $N^n(R)$ is a normalised program.*

Proof. (outline)

Let $t_1, t_2, \ldots, t_n$ be a maximal dependency path in R. Suppose $D(t_i, t_j)$ holds for some $i < j$.

- If $t_i \subseteq t_j$, then in $N(R)$ the above maximal dependency path is replaced by a shorter maximal dependency path $t_1, t_2, \ldots, t_{j-1}$ (since t_j was replaced by t_i).
- If $t_i \not\subseteq t_j$ then $N(R)$ contains a definition of $t_i \sqcup t_j$. It can be shown that in $N(R)$ $t_1, t_2, \ldots, t_n$ is replaced by a corresponding maximal dependency path of length n, which contains predicates whose procedure definitions contain more functions in their clause heads than the predicates definitions in $t_1, t_2, \ldots, t_n$.

Since the number of function symbols in procedures has a finite upper bound and the length of maximal dependency chains has a finite lower bound, the number of possible applications of normalisation is finite.

□

Definition 3.10 *Let R be an RUL program. Then* $\mathbf{norm}(R) = N^n(R)$ *such that $N^n(R)$ is normalised.*

The next lemma shows that normalisation increases the size of the success set of any predicate that occurs in both R and $N(R)$.

Lemma 3.11 *Let R be an RUL program containing a predicate t such that t occurs both in R and $N(R)$. Then $success_R(t) \subseteq success_{N(R)}(t)$.*

Proof. (outline)

During normalisation, whenever an occurrence of a predicate s is replaced by another predicate q, it is the case that $success_R(s) \subseteq success_{N(R)}(q)$, since either $s \subseteq q$ or $q = s \sqcup r$. Hence any answers obtained for s are also obtained for q. t either depends on s or it does not: in either case the property holds.

□

The next lemma shows that normalised RUL programs containing a fixed finite set of function symbols can express only a finite number of distinct sets of terms. This property is later used to show the termination of the algorithm for deriving a regular approximation of a program.

Lemma 3.12 *Let $F = \{f_1/n_1, \ldots, f_m/n_m\}$ be a finite number of function symbols. Then there is a finite number of different sets of the form $success_R(t)$, where R contains only function symbols from F.*

PROOF.

Let m be the number of distinct functions in F. Then there are $2^m - 1$ different non-empty sets of functions that can appear in the head of a procedure in R. Since R is normalised, there are no t and s such that $D(t, s)$. Hence the maximum length of maximal dependency paths is $2^m - 1$.

The proof is then by induction on the length d of maximal dependency paths in R.

> Basis: $d = 0$. In this case the procedures in R consist either of unit clauses of the form $t(a) \leftarrow true$ or clauses of the form $t(f(x)) \leftarrow t(x)$. There is a finite number of sets $success_R(t)$ obtainable from definitions of t of this form.
>
> Assume that the result holds for maximal dependency paths of length at most d. Then we show it holds for maximal dependency paths of length $d + 1$. Suppose t is at the start of a maximal dependency paths of length $d+1$. Then the procedure for t consists of clauses of the form $t(f(x_1, \ldots, x_k)) \leftarrow t_1(x_1), \ldots, t_k(x_k)$. Each t_j starts a maximal dependency paths of length at most d. Hence by the induction hypothesis there is a finite number of different possible sets $success_R(t_j)$, $(1 \leq j \leq k)$. By permuting all these different versions there is a finite number of possible versions of the clause $t(f(x_1, \ldots, x_k)) \leftarrow t_1(x_1), \ldots, t_k(x_k)$. Hence there is a finite number of distinct sets $success_R(t)$.

Thus for any finite upper bound on maximal dependency chains in R, there is a finite number of distinct definable sets $success_R(t)$.

□

Let Q be a set of regular clauses, where Q is not an RUL program. That is, there are two clauses which have the same predicate and function symbol in their heads. An RUL program can be derived from Q by the following operation.

Definition 3.13 *Let Q be a set of regular unary clauses and let there be two clauses*

- $t(f(x_1, \ldots, x_n)) \leftarrow s_1(x_1), \ldots, s_n(x_n)$ *and*
- $t(f(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n)$

such that $s_1, \ldots, s_n, r_1, \ldots, r_n$ are defined by a subset of Q that is an RUL program. Then replace the two clauses by

- $t(f(x_1, \ldots, x_n)) \leftarrow q_1(x_1), \ldots, q_n(x_n)$

where $q_i = r_i \sqcup s_i$, $(i \leq i \leq n)$, and add the definitions of q_i to Q.

Repeat this operation until it cannot be applied any more. The final result is called $\mathbf{reg}(Q)$.

As defined, $\mathbf{reg}(Q)$ is not an RUL program for all Q, but for the purposes of this paper $\mathbf{reg}(Q)$ is an RUL program since it is applied only to regularise the procedure *approx*.

4 Deriving a Regular Approximation of a Normal Program

In this section an algorithm is developed for computing a regular approximation of a given logic program. One of the central ideas is to solve a definite goal with respect to a set of regular definitions of predicates.

Let $\leftarrow A_1, \ldots, A_n$ be a definite goal. The solution of this goal with respect to a set of regular definitions is next defined.

Let $D \cup R$ be regular definitions of the predicates in $A_1, \ldots, A_n$ where D consists of clauses $approx(p^i(x_1, \ldots, x_{n^i})) \leftarrow B$. When the resolution of a goal with a clause in D is mentioned, the clause $p^i(x_1, \ldots, x_{n^i}) \leftarrow B$ is used. Then the solution (if it exists) for $\leftarrow A_1, \ldots, A_n$ in $D \cup R$ is obtained in two stages:

1. Resolve the goal $\leftarrow A_1, \ldots, A_n$ with clauses in $D \cup R$ until the resolvent is of the form $\leftarrow t_1(y_1), \ldots, t_k(y_m)$. That is, the resolvent contains only unary predicates, whose arguments are variables.

2. If some variable y occurs more than once in the resolvent, say as an argument to t_r and t_s, then compute $t = t_r \cap t_s$, delete $t_r(y)$ and $t_s(y)$ from the resolvent, and add $t(y)$ instead. Repeat this operation until each variable occurs exactly once in the resolvent.

Note that this procedure could fail either because resolution fails, or because some intersection is empty. If it does not fail, the procedure yields a unique result.

The solution of the goal $\leftarrow A_1, \ldots, A_n$ is then the final resolvent, with associated RUL program $R \cup R'$, where R' is the collection of definitions of intersected predicates.

4.1 Regular Solution of a Definite Clause

Let $A \leftarrow B$ be a definite clause. Let $D \cup R$ be a regular definition of the predicates in B. Obtain the solution of B as described above, namely, a resolvent $\leftarrow s_1(y_1), \ldots, s_m(y_m)$, where $y_1, \ldots, y_m$ are distinct variables, where the predicates $s_1, \ldots, s_m$ are defined by an RUL program $R \cup R'$.

The solution of the clause at this stage is $A \leftarrow s_1(y_1), \ldots, s_m(y_m)$. Transform it to $approx(A) \leftarrow s_1(y_1), \ldots, s_m(y_m)$. The arguments of A can now be transformed to a regular representation in the following steps. Let $A = p(u_1, \ldots, u_k)$.

1. For each variable z in A that does not occur in B add an atom $any(z)$ to the body $s_1(y_1), \ldots, s_m(y_m)$.

2. For each variable z that occurs more than once in A, where $t(z)$ occurs in the body, replace one occurrence of z in A by a fresh variable w and add $t(w)$ to the body. Repeat until each variable occurs once only.

3. For each argument u_j of A, perform the following:

   (a) If u_j is a variable, do nothing.

(b) If u_j is a non-variable $f(v_1, \ldots, v_q)$, introduce a new unary predicate t and a new variable z, add $t(z)$ to the body and remove $r(y)$ from the body for every y in u_j. Let the conjunction of the atoms $r(y)$ removed be Q. Form a clause $t(f(v_1, \ldots, v_q)) \leftarrow Q$. Apply step (3) to this clause.

At the end of the procedure a regular definition for p/k, the predicate of A, is obtained, along with a set of RUL clauses defining the subsidiary clauses that were introduced.

Definition 4.1 **solve**$(A \leftarrow B, R)$

Let $A \leftarrow B$ be a definite clause, and R be a regular definition of predicates in B. The operation **solve**$(A \leftarrow B, R)$ *is defined to be $R \cup Q$, where Q is the set of all the regular clauses introduced by the solution of $A \leftarrow B$ in R described above.*

Note that **solve**$(A \leftarrow B, R)$ is not necessarily an RUL program, since it may contain two clauses with heads $approx(p(x_1, \ldots, x_k))$, one from R and one derived in the solution process.

Example 2 *Let the clause be*

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

and let $R = \{approx(append(x, y, z)) \leftarrow t_1(x), any(y), any(z), t_1([]) \leftarrow true\}$.

The solution of the body yields $append([x|xs], ys, [x|zs]) \leftarrow t_1(xs), any(ys), any(zs)$. *The transformation of the head yields* $approx(append(x, y, z)) \leftarrow t_2(x), any(y), t_3(z)$, *with additional RUL clauses* $\{t_2([x|xs]) \leftarrow any(x), t_1(xs), t_3([z|zs]) \leftarrow any(z), any(zs)\}$.

So **solve**$(append([x|xs], ys, [x|zs]) \leftarrow append(xs, ys, zs), R)$ *yields the following program.*

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

4.2 Approximation of a Definite Program

Let P be a definite program. Given a set of regular definitions of the predicates in P, a solution for each clause in P can be computed, as shown above. For each clause defining p/n in P, one regular definition is derived.

A function T_P^R is now defined for a definite program P. The name of the function is derived from the T_P function for computing the minimal model of a definite program [13].

Definition 4.2 *the function* T_P^R

Let P be a definite program. Let D be a regular definition of predicates in P.

A function $T_P^R(D)$ *is defined as follows:*

$$T_P^R(D) = \mathbf{norm}(\mathbf{reg}(\bigcup \left\{ \mathbf{solve}(A \leftarrow B, D) \;\middle|\; A \leftarrow B \in P \right\}))$$

If we restrict attention to sets of regular unary clauses that are either empty or contain a predicate *approx*, and the only other predicates are those that *approx* depends on, then a partial order $\sqsubseteq$ is defined on such RUL programs D by $D_1 \sqsubseteq D_2$ iff $success_{D_1}(approx) \subseteq success_{D_2}(approx)$.
CLAIM. The operations **reg**, **norm**, $\cup$ and **solve** are monotonic with respect to this partial order on D.

Hence T_P^R is monotonic, and thus it has a least fixed point F in which $success_F(approx) = success_{T_P^R(F)}(approx)$.

Lemma 4.3 *Let P be a definite program and let F be a fixed point of T_P^R. Then F is a regular approximation of P.*

PROOF. (outline)
The proof first shows that T_P^R approximates the usual semantic function T_P, in the following sense. Let D be a regular definition of predicates in P. Then $success_D(approx)$ is a Herbrand interpretation of P. It can be shown that $T_P(success_F(approx)) \subseteq T_P^R(F)$.
By induction on the number of iterations of T_P and T_P^R it can be shown that $T_P^n(\emptyset) \subseteq T_P^{R^n}(\emptyset)$ and hence this holds at the fixed point of both operators. The lemma follows immediately. □

The fixed point of T_P^R is found in a finite number of iterations, because of the property proved in Lemma 3.12.

4.3 Regular Approximation of Normal Programs

The approximation procedure can be applied to normal programs as follows:

1. Let P be a normal program. Let P' be the program obtained by deleting all negative literals from P. P' is clearly a definite program that is a safe approximation of P.
2. Compute a regular approximation of P'.

4.4 Regular Approximation of Prolog Programs

Prolog built-in predicates can be approximated by assigning *any* to each argument. More precise approximation, such as *integer* could be defined.
The procedure can safely be applied to any program that does not use *assert* and *retract* and related predicates, which can obviously dynamically alter the success set of a program.

5 Implementation and Applications

A procedure to compute the least fixed point of T_P^R has been implemented in a straightforward naive way, using the algorithm below (where `T` represents T_P^R):

```
BEGIN
     i := 0;
     F[0] := {};
     S[0] := {};
     REPEAT
          F[i+1] := T(F[i]);
          S[i+1] := success[F[i+1]](approx);
          i := i+1
     UNTIL S[i] = S[i-1]
END
```

The value of `F[i]` at the end is the safe approximation program. There are well-known ways to improve the efficiency of fixed point computations. A method is discussed in [10]. We have so far implemented only the above naive algorithm. The remainder of this section contains discussion of applications of safe approximations.

5.1 Relation to Type Inference

A regular approximation of a program bears a superficial resemblance to type information about the program, and indeed most of the previous work on regular structures in logic programming has been done in connection with type systems for logic programs [18], [8], [15].

However regular approximations should not be confused with types from the semantic point of view, although some connections between types and RUL programs could be made. The intention of providing types is not to describe a superset of the program's success set; it is rather to prescribe the set of well-typed programs, which is in general not a superset of the success set (of the corresponding untyped program). RUL programs could also be used in a prescriptive fashion, as in [18], but they do not provide the equivalent of a typed interpretation of a program.

5.2 Static Analysis for Debugging

From the practical standpoint, regular approximations serve some of the same purposes as types, namely, pinning down the intended interpretation of a program.

Given a program P, a regular approximation F can be computed and used to do static debugging. The following points indicate the kinds of analysis that can be done.

- The operator **solve**$(A \leftarrow B, F)$ can be evaluated for each clause $A \leftarrow B$ in P. If for some clause the result is empty, the clause is redundant, and indicates a bug.
- The regular definitions in F can be examined to see whether they give supersets of the intended success sets for each program predicate.
- If an RUL program D is used to prescribe the structure of the success set of a program (as in [18]) then D can be compared with F. Also, the operator **solve**(C, D) can be computed for each clause C. If any clause in P has an empty solution in D, then that

clause is redundant with respect to the prescribed interpretation. If $T_P^R(D) \subseteq D$, then it indicates that a more precise prescription of D could be given. If $D \subseteq T_P^R(D)$ then D may not cover all the results obtainable from P, so either D or P may be faulty.

These uses are similar to those presented in [18].

5.3 Regular Approximations of Call-Answer Programs

The oddly-named "magic set" and "Alexander" transformations were introduced as recursive query evaluation techniques [2], [3] and [16]. They have since been adapted for use in program analysis [5], [7], [12], [14] and [17].

The motivation for using the transformations is the following: we want to examine particular computations rather than the model of the program. If we analyse a program (as it is) we get the success set of the program, which is in general larger than the information we want for a particular call to the program. By transforming the program with the magic set transformation, we can restrict attention to certain calls to predicates rather than their general models. This analysis information will in general be more precise and therefore more useful. This may be the case even when the supplied call to a predicate is unrestricted, since calls from within clause bodies may be instantiated.

The intuitive idea behind the method will be explained with the following simple program:

$p(x) \leftarrow q(x), r(x).$
$q(a).$
$q(b).$
$r(a).$

Let us for the sake of this example assume that we want information about the arguments with which p is called.

We now reason as follows: in the first clause, assuming a left-to-right computation rule, for $r(x)$ to be called, $q(x)$ must succeed and $p(x)$ must be called. For $q(x)$ to succeed, $q(x)$ must be called , and for $q(x)$ to be called, $p(x)$ must be called. This information about the calls and answers of atoms in the program can be expressed in the following transformed program:

$answer(p(x)) \leftarrow$
$\quad call(p(x)),$
$\quad answer(q(x)),$
$\quad answer(r(x)).$
$answer(q(a)) \leftarrow$
$\quad call(q(a)).$
$answer(q(b)) \leftarrow$
$\quad call(q(b)).$
$answer(r(a)) \leftarrow$
$\quad call(r(a)).$
$call(r(x)) \leftarrow$
$\quad call(p(x)),$

$answer(q(x))$.
$call(q(x)) \leftarrow$
$call(p(x))$.

We also add the information that $p(x)$ can be called (the top level query) to the above program by adding the clause:

$call(p(x))$.

We assume that this is the only call to the program. The general form of the above transformation can be described by the following meta-interpreter.

$call(b_j) \leftarrow clause(h \leftarrow bs),$
$conj(bs_1, (b_j, -), bs),$
$answer(bs_1),$
$call(h).$

$answer(true).$
$answer((b, bs)) \leftarrow answer(b),$
$answer(bs).$
$answer(b) \leftarrow b \neq (-,-), b \neq true,$
$clause(b \leftarrow c),$
$call(b),$
$answer(c).$

where $conj(p, q, r)$ is true if the conjunction of p and q is r.

Note that the calls and answers produced are not in general identical to those arising in SLD computations, since some spurious instances of calls and answers may be produced. But since we are interested in safe approximation of the ground success set this does not matter.

5.3.1 Analysis of Call-Answer Approximations

The use of a "magic set" style transformation can give more precise approximations, when computing regular approximations. The approximation of the *answer* predicate can often be used in place of the approximation of the program itself, in static analysis for debugging, as discussed in Section 5.2.

As an illustration, consider the naive reverse program

Example 3 *Let P be the program:*

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

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

By using the procedure for regular approximation, the following program was obtained by our implementation:

$reverse(x1, x2) \leftarrow t53(x1), any(x2)$

$t53([]) \leftarrow true$
$t53([x1|x2]) \leftarrow any(x1), t53(x2)$

$append(x1, x2, x3) \leftarrow t61(x1), any(x2), any(x3)$

$t61([]) \leftarrow true$
$t61([x1|x2]) \leftarrow any(x1), t61(x2)$

As can be seen, the approximation loses all information about the second argument of *reverse*. The reason is that since the base case of *append* allows any term for the second two arguments of *append*, this propagates into the second argument of *reverse*.

When $reverse(x, y)$ is computed, the only calls to *append* that arise have the second argument instantiated as a singleton list. This restricts the possible answers to *append* and hence to *reverse*. This can be captured using a call-answer transformation of the program, with the call to $reverse(x, y)$. The transformed program is as follows.

$answer(reverse([], [])) \leftarrow$
$\quad call(reverse([], [])).$
$answer(reverse([x|xs], ys)) \leftarrow$
$\quad call(reverse([x|xs], ys)),$
$\quad answer(reverse(xs, zs)),$
$\quad answer(append(zs, [x], ys)).$

$answer(append([], ys, ys)) \leftarrow$
$\quad call(append([], ys, ys)).$
$answer(append([x|xs], ys, [x|zs])) \leftarrow$
$\quad call(append([x|xs], ys, [x|zs])),$
$\quad answer(append(xs, ys, zs)).$

$call(reverse(x, y)).$
$call(reverse(xs, \_) \leftarrow$
$\quad call(reverse([\_|xs], \_).$
$call(append(zs, [x], ys)) \leftarrow$
$\quad call(reverse([x|xs], ys)),$
$\quad answer(reverse(xs, zs)).$
$call(append(xs, ys, zs)) \leftarrow$
$\quad call(append([x|xs], ys, [x|zs])).$

The regular approximation of this program returned by our procedure is:

$answer(reverse(x1, x2)) : -t172(x1), t165(x2)$

$t172([]) : -true$
$t172([x1|x2]) : -any(x1), t172(x2)$

$t165([x1|x2]) : -any(x1), t165(x2)$
$t165([]) : -true$

$answer(append(x1, x2, x3)) : -t180(x1), t139(x2), t188(x3)$

$t180([]) : -true$
$t180([x1|x2]) : -any(x1), t180(x2)$
$t139([x1|x2]) : -any(x1), t140(x2)$
$t188([x1|x2]) : -any(x1), t188(x2)$
$t188([]) : -true$
$t140([]) : -true$

$call(reverse(x1, x2)) : -any(x1), any(x2)$

$call(append(x1, x2, x3)) : -t96(x1), t139(x2), any(x3)$

$t96([x1|x2]) : -any(x1), t96(x2)$
$t96([]) : -true$
$t139([x1|x2]) : -any(x1), t140(x2)$
$t140([]) : -true$

As can be seen, the approximations for the answers of $reverse(x, y)$ are now more precise, since both arguments consist of lists. Also, the answers for *append* show that the third argument is a list and that the second argument is a singleton list.

5.4 Specialisation Using Regular Approximation

A more interesting use of the transformations is to analyse particular computations of programs, rather than the success set of the whole program. Given a program P and atomic goal $\leftarrow A$, the magic transformation of P, along with the clause $call(A) \leftarrow true$ can be analysed. This gives information about the calls and answers generated during the computation of $P \cup \{\leftarrow A\}$. From this information, an optimised version of P can be derived (e.g. some clauses in P may not be needed to compute answers for A). Our current research is to integrate regular approximations in an algorithm for program specialisation [9], and analyse the behaviour of constraint interpreters. The method of integration is described in [6].

5.5 Machine Learning

Regular approximation of a program can be regarded as an example of inductive learning, in which the regular definitions are generalisations of the information given in the program.

The literature on machine learning includes procedures for inferring regular grammars from given examples of strings of a language. Biermann [4] surveys some of these methods and gives an algorithm for performing this kind of learning.

We performed some small experiments to see whether such methods could be reproduced using our regular approximation derivation. In fact we were able to achieve very similar results to those given in [4]. The following example is given by Biermann.

Given a set of sample strings $\{a, aa, ba, aaa, aba, abb, baa, bba\}$ the problem is to derive a grammar for a language including these strings. First a corresponding logic program is written.

$l(a) \leftarrow true$
$l(a(a)) \leftarrow true$
$l(b(a)) \leftarrow true$
$l(b(b)) \leftarrow true$
$l(a(a(a))) \leftarrow true$
$l(a(b(a))) \leftarrow true$
$l(a(b(b))) \leftarrow true$
$l(b(a(a))) \leftarrow true$
$l(b(b(a))) \leftarrow true$

In this representation the predicate $l(x)$ states that x is in the language, and the functors $a/0$ and $b/0$ denote singleton strings a and b, and the functors $a/1$ and $b/1$ denote strings starting with a and b respectively.

Applying our procedure on this program, we obtained this approximation. Since the program contains only unit clauses, the first iteration gives the fixed point.

$l(x1) : -t157(x1)$

$t157(b(x1)) : -t157(x1)$
$t157(a(x1)) : -t143(x1)$
$t157(a) : -true$
$t157(b) : -true$

$t143(b(x1)) : -t145(x1)$
$t143(a) : -true$
$t143(a(x1)) : -t98(x1)$

$t145(a) : -true$
$t145(b) : -true$

$t98(a) : -true$

This is not quite identical to Biermann's result. The difference amounts to a different normalisation procedure than the one described in this paper. However, Biermann discusses the use of different methods of constructing grammars, corresponding to variations on the normalisation.

The following alternative definition of $D(t, s)$, for instance, leads to a different normalisation and hence a different approximation.

Definition 5.1 $D'(t, s)$

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 $s \subseteq t$.

$D'(t,s)$ would not be suitable for the general case since it does not give a monotonic operator **norm** as required in Section 4. However it is fine for unit clause programs and its use on the the above example gives something equivalent to Biermann's result, namely:

$$l(x1) : -t152(x1)$$

$$t152(b(x1)) : -t154(x1)$$
$$t152(a(x1)) : -t152(x1)$$
$$t152(a) : -true$$

$$t154(b(x1)) : -t152(x1)$$
$$t154(a(x1)) : -t152(x1)$$
$$t154(a) : -true$$
$$t154(b) : -true$$

Several other learning algorithms produce a finite automaton as a result, and since regular languages and deterministic finite automata define the same class of languages, it suggests that our method of regular approximation could be applied to other learning problems. We are currently attempting to reproduce Biermann's algorithm for learning Turing machines from program traces. Again, the method would represent the traces as clauses in a logic program, and then compute an approximation that could generate the traces.

6 Related Work and Conclusion

The procedure in this paper can be compared to methods for inferring regular descriptions or "types" for the predicates of logic programs [19], [15], [8]. Our method differs in that it is a bottom-up fixed point computation. This makes it more precise for some programs, such as looping definitions. For instance, the program $\{p(x) \leftarrow p(x)\}$ would get an empty approximation using the procedure in this paper, whereas apparently other methods would give an approximation $\{p(x) \leftarrow any(x)\}$.

Similarly, a program such as $\{p(f(f(a))) \leftarrow true, p(x) \leftarrow p(f(x))\}$ would be assigned an approximation $\{p(x) \leftarrow any(x)\}$ by other methods, whereas ours gives $\{p(x) \leftarrow t(x), t(a) \leftarrow true, t(f(x)) \leftarrow t(x)\}$. Such differences are important when applying the approximation to call-answer programs (Section 5.3) since the base case of a call predicate may contain structure which is successively broken down during the computation.

Although [18] contains a definition of an abstraction of the T_P operator in order to define a type, similar in spirit to T_P^R in this paper, it could not used as a means of constructing an approximation.

Other methods such as those in [15] and [19] include polymorphism, which our method does not. This gives extra precision where we would obtain $any(x)$ in some cases, since

polymorphism captures structure sharing information that cannot be represented in regular structures.

The fixed point computation can be expensive for larger programs, and further study on the complexity is needed. Our preliminary naive implementation is written in Prolog and runs on a SPARCstation-2. On a call-answer transformed version of a unification algorithm for a ground representation, which contains approximately 80 clauses, the procedure takes several minutes to reach a fixed point.

Further research will be aimed at incorporating the regular approximation procedure into a program specialisation system [9]. By getting more precise descriptions of call and answer patterns during partial evaluation, more specialised programs can be obtained.

Acknowledgements

Thanks to Ian Holyer, Bristol University, for useful discussions about regular programs.

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] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6^{th} ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, California*, 1987.

[4] 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.

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

[6] D.A. de Waal and J. Gallagher. Specialisation of a unification algorithm. Technical Report TR-91-14, University of Bristol, July 1991. (To appear in proceedings of LOP-STR'91).

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

[8] T. Frühwirth. Type inference by program transformation and partial evaluation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, 1988.

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

[10] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 6(2), 1988.

[11] 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.

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

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

[14] C.S. Mellish. Using specialisation to reconstruct two mode inference systems. Technical report, University of Edinburgh, March 1990.

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

[16] 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.

[17] 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.

[18] E. Yardeni and E.Y. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*. MIT Press, 1978.

[19] J. Zobel. Derivation of polymorphic types for prolog programs. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*. MIT Press, 1988.