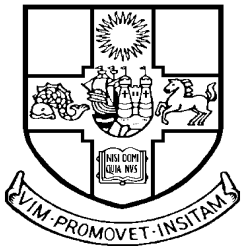
Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions

Huseyin Saglam John P. Gallagher

July 1995 **CSTR-95-017**



University of Bristol

Department of Computer Science

Also issued as ACRC-95:CS-017

Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions [1]

Hüseyin Sağlam **John P. Gallagher**

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

e-mail: saglam@compsci.bristol.ac.uk, john@compsci.bristol.ac.uk
fax: +44-117-9251154

Abstract

Approximate descriptions of the success set of a program have many uses in program development and optimisation. For untyped logic programming languages, regular approximation is a practical and useful tool. In this paper we consider the problem of approximating the meaning of programs in which some (polymorphic) type information is given. This situation can arise in constraint logic programming languages. In untyped languages the user could impose types on selected symbols. Even in strongly typed languages we may be able to derive more precise descriptions of the meaning, or to consider restricted uses of polymorphic typed predicates. We propose a practical two-stage method: first the original program is transformed by replacing typed arguments by corresponding polymorphic type terms. For well-typed programs the resulting program is an abstraction of the original. Second, an established algorithm for regular approximation is applied to the transformed program. The algorithm is guaranteed to terminate without using artificial techniques such as depth-k bounds on (type) terms. The derived description combines polymorphic type terms, including union types, with regular descriptions of untyped terms. The method allows goal-dependent analysis as well as goal-independent analysis of a complete program. We show some experimental results demonstrating the speed and precision of the method and show that it scales up well when applied to larger programs.

Keywords: abstract interpretation, type inference, regular approximation, polymorphic types, type union.

1 Introduction

Approximate descriptions of the success set of a program have many uses in program development and optimisation. Abstract interpretation is a framework for inferring descriptions of various kinds (for example descriptions including modes, sharing and regular term information). Regular approximation is an abstract interpretation sometimes identified with "type inference".

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

Type systems are designed to allow the programmer to describe the set of intended values that an argument can take. Types are semantic objects associated with the domain of interpretation (*prescriptive types*), whereas regular approximations are syntactic and describe the success set of programs (*descriptive types*). However for definite programs these two notions coincide quite closely for practical purposes. In this paper we consider the interaction between type information and regular term descriptions inferred by abstract interpretation. Our aim is to infer an approximation of the success set of partly-typed programs by means of a mixture of type terms and regular descriptions. We use a two-stage method to achieve this.

In the next section type systems and the notion of type inference are introduced. Section 3 briefly describes abstract interpretation. In Section 4 well-typed CLP programs and the correctness of the approximation is examined. An algorithm for regular approximation using regular unary logic programs is given in Section 5. Section 6 introduces the two-stage method for computing an approximation of a partly-typed program. Some experimental results are given in Section 7. Other approaches to the problem are discussed in Section 8. Finally Section 9 concludes.

2 Type Systems

Given a set of constants including $\bot$, called *base types*, a set of term *constructors* with given arities, including a binary constructor $\cup$, and a set of variables, a *type term* (or simply a *type*) is either a base type, a variable or a constructor of arity n applied to n type terms. Let τ_1 and τ_2 be types. Then τ_1 is an *instance* of τ_2 if τ_1 can be obtained from τ_2 by substituting type variables in τ_2 by some other types. Two types τ_1 and τ_2 are *equivalent* if τ_1 is an *instance* of τ_2 and τ_2 is an *instance* of τ_1, that is, they are equal up to renaming of type variables. Given two types τ_1 and τ_2 the term $\tau_1 \cup \tau_2$ is called their *union*. It describes the smallest type containing both the terms of type τ_1 and the terms of type τ_2. The type $\bot$ is called the *undefined* type. The meaning of $type\_name(\bot, ..., \bot)$ is that a type associated with a constant of a parametric type is partially undefined. The role of $\bot$ as the identity for the union operation is given later in this section.

2.1 Type Expressions

Given a set of variables V, constants K and functions F in a language L, the set of terms constructible from V, K and F is written as $Term(V, K, F)$. Type expressions are in a language which is an extension of the language underlying L. Let B be a set of base types, K be a set of constants, F be a set of function symbols and C be a set of type constuctors. We assume that $K \cap B = \emptyset$ and $F \cap C = \emptyset$. Then we define $Term_L = Term(V, K, F)$ and $Term_L^\tau = Term(V, K \cup B, F \cup C)$. In other words *type expressions* are the set of terms constructible from V, K, B, F and C.

2.2 Type declarations

Type declarations are used to associate types with the expressions of a language L containing a set of constants K, function symbols F and variables V. Informally, each type τ is associated with a set of terms in L. If t is in the set of terms associated with τ we say that t *is of type*

τ. A parametric type is a type constructor applied to types ζ_i. *Polymorphism* results from the possibility of having type variables in the type terms. A *parametric* type declaration for an n-ary type (possibly $n = 0$) has the following form.

$$
\begin{array}{lcl}
c & \rightarrow & type\_name(\perp, ..., \perp) \\
... & & \\
f(\tau_1, ..., \tau_m) & \rightarrow & type\_name(\zeta_1, ..., \zeta_n)
\end{array}
$$

Each line $u \rightarrow v$ is called a *rewrite rule*. The left part of each rule is either a constant or the application of a function symbol in F to types τ_i. The right part of each rule is the application of the type to types ζ_i. We impose the condition that all type variables on the right hand side of a rule also occur on the left. We normally assume that the type of a constant c of some parametric type is given as $c \rightarrow type\_name(\perp, ..., \perp)$. The type declarations, for example, for the type **num** and the type **list** can be given as follows.

```
0 --> num   1 --> num    2 --> num   . . .    num + num --> num
[ ] --> list( bottom )
[ X | list(Y) ] --> list(union(X,Y))
```

In addition to these type declarations, we introduce another set of rewrite rules to handle union type operations, which consist of the set of the rules

$$
\begin{array}{l}
\{\ x \cup x \rightarrow x, \\
\quad x \cup \perp \rightarrow x, \\
\quad \perp \cup x \rightarrow x\ \}.
\end{array}
$$

The types of n-ary predicate symbols can also be declared and are of the form $p(\tau_1,...,\tau_n)$ where τ_i are types and p is an n-ary predicate symbol.

Union types can be used to describe *heterogeneous* structures in a language such as lists or tuples containing elements of different types. By contrast, a *homogeneous* structure is one containing elements of one type only.

2.3 Type Reduction

Given a language L and a set of type declarations S for L, let RR be the set of rewrite rules occurring in S, and let t be a term in L. A determinate redex in t is a subterm of t which unifies with exactly one left hand side of a rule in RR (with variables in the rule and the variables in t standardised apart). We are assuming that more than one rewrite rule exists, otherwise every redex is determinate.

Let t be a term and r some determinate redex in t. Let $u \rightarrow v$ be the rule matching r, where $mgu(r, u) = \theta$. Then t is reduced by replacing the occurrence of r by v and applying θ to the result. Type reduction of t continues until no more (determinate) reduction is possible.

Example 1 *Consider the term* `[1|L]` *with the reduction rules*

```
[X|list(Y)] --> list(union(X,Y))
1 --> num
```

Type reduction of `[1|L]` *does not depend on the order of reduction and* `1` *is one of the determinate redexes in t. The first reduction rule applied is* `1 --> num (u = 1, v = num)`*.* $mgu(r,u) = \theta = mgu(1,1) = \epsilon$*. The occurence of* `1` *is replaced by* `num`*, and* $\theta = \epsilon$ *is applied to the result.*

Now we have the determinate redex `[num|L]` *matched by the first reduction rule where* $mgu($`[num|L], [X|list(Y)]`$) = \{$`X/num, L/list(Y)`$\}$*. The occurence of the determinate redex* `[num|L]` *is replaced by* `list(union(X,Y))`*. After applying* $\theta = \{$`X/num, L/list(Y)`$\}$ *to the result* `list(union(num,Y))` *is obtained and no more reduction is possible.*

We define a function $\mathcal{R} : Term_L \rightarrow Term_L^\tau$ as $\mathcal{R}(t) = T$, where T cannot be further reduced. The function $\mathcal{R}$ extends naturally to apply to atoms and clauses. It can be shown that $\mathcal{R}$ terminates and yields a unique result not depending on the order of reduction. Type reduction is related to "narrowing".

Definition 2.1 *well-typed expression*

Let E be some expression. Let RR be a set of rewrite rules corresponding to some type declarations. E is **well-typed** *with respect to RR if, for all subterms t of E,* $\mathcal{R}(t)$ *contains no function symbol occurring on the left-hand-side of a rule in RR.*

2.4 Type Inference

The process of determining the type of program units is called *type inference*. The programmer might not declare types for every symbol in a program, and almost never has to declare the types of variables. We are interested in cases where the programmer (or the language implementation) defines the type of certain constants, function symbols and predicates, and the types of all other symbols has to be inferred.

Two aspects of type inference should be distinguished. One is that if some type declarations have been given, the system checks whether those declarations are respected, and infers type descriptions for other terms as far as possible. This is called *prescriptive typing*. The second aspect is that for predicates where no type restrictions are imposed, the "type" that is inferred is a description of a set of atoms such that the success set of the predicate is a subset of the inferred description. This is called *descriptive typing*. The programmer can then compare this description with the intended (but not declared) type of that predicate. Ideally, the intended type should include the success set, in other words, the program should only succeed for "well-typed" atomic goals. Thus the word "type inference" is not strictly correct although for definite logic programs the concept of "type" and success set of predicates coincide quite closely.

3 Abstract Interpretation

Let us assume that a program P's computational behaviour is fully characterised as the least fixed point (*lfp*) of a continuous function $E_P : D \rightarrow D$, where $(D, \leq_D)$ is a lattice with

partial order $\leq_D$. Elements of D are program meanings. Thus we can state that $(D, \leq_D, E_P)$ defines a *concrete fixpoint semantics* for the program P. Since computations can be infinite the above definition cannot be used for static program analysis. Therefore a *non-standard fixpoint semantics* based on an abstract lattice $(A, \leq_A)$ is used. An abstract semantic function $\mathcal{E}_P : A \to A$ is defined whose least fixed point is the abstract meaning of the program P. If $(A, \leq_A)$ has finite height then the analysis is finitely computable (though finiteness is not a necessary condition).

The concrete and the abstract semantics are related by defining a pair of functions α and γ *(abstraction* and *concretisation* functions respectively), which form a *Galois insertion* between $(A, \leq_A)$ and $(D, \leq_D)$ [7].

To summarise, an abstract interpretation scheme for a program is a tuple

$$\langle (D, \leq_D), E_P, (A, \leq_A), \mathcal{E}_P, \alpha, \gamma \rangle$$

such that:

- $(A, \leq_A)$ and $(D, \leq_D)$ are complete lattices,
- $E_P : D \to D$ and $\mathcal{E}_P : A \to A$ are monotonic functions,
- $\alpha : D \to A$ (abstraction) and $\gamma : A \to D$ (concretisation) are monotonic functions,
- $\forall a \in A$, $\alpha(\gamma(a)) = a$,
- $\forall d \in D$, $d \leq_D \gamma(\alpha(d))$

According to the last two conditions the concretisation cannot cause any loss of information, while the abstraction of a concrete object may cause some loss of information.

The correctness, or safety of an abstract interpretation is formally expressed by the requirement that $lfp(E_P) \leq_D \gamma(lfp(\mathcal{E}_P))$. In other words, the abstract meaning of P represents (via γ) a greater program meaning (a safe approximation) than the concrete meaning of P.

4 Well-typed CLP Programs

A program is well-typed if all its clauses are well-typed. Note that the condition that all subterms are well-typed is necessary because rewrite rules are not in general *transparent*; that is, variables on the left-hand-side of a rewrite rule might not appear on the right. A badly-typed argument could therefore disappear during reduction.

A clause in a constraint logic program (CLP) is written $H \leftarrow B, C$ where B are atoms with user-defined predicates and C is a conjunction of constraints. Given a constraint logic program P, we apply $\mathcal{R}$ to each clause in P, yielding a program P_τ, called the *type-reduced* version of P.

The program obtained by applying type reduction to each clause of a program is a "safe approximation" of the original program, in the sense to be explained below. The notion of safety depends on the notion of "well-typed".

Let P be the original constraint logic program; its success set $SS[P]$ is defined as follows. The notation $ground(P)$ means the set of ground instances of clauses in P. We assume a fixed interpretation of the constraint predicates.

Definition 4.1 *Let P be a CLP program. Its k-success set $SS^k[P]$ is defined inductively as*

$$SS^1[P] = \left\{ p(\bar{t}) \;\middle|\; p(\bar{s}) \leftarrow C \in ground(P), C \text{ is a true constraint}, p(\bar{s}) \text{ is well typed}, \; \bar{t} = \bar{s} \right\}$$

$$SS^k[P] = \left\{ p(\bar{t}) \;\middle|\; \begin{array}{l} p(\bar{s}) \leftarrow B_1, \ldots, B_n, C \in ground(P), \\ p(\bar{s}) \text{ is well typed}, C \text{ is a true constraint}, \\ \{B_1, \ldots, B_n\} \subseteq SS^{k-1}[P], \\ \bar{t} = \bar{s} \end{array} \right\}$$

Here $=$ means the equality theory underlying the constraint program.

$$SS[P] = \bigcup_{k=1}^{\infty} SS^k[P]$$

Let P_τ be the type-reduced version of P. In P_τ we assume that equality is free equality (syntactic equality) and that the constraint predicates are defined by abstract "facts" of the form $p(\tau_1, \ldots, \tau_n) \leftarrow true$ where p/n is a constraint predicate and $(\tau_1, \ldots, \tau_n)$ is its declared type.

In the following definition, the typed constraint predicates in type-reduced clauses are evaluated with respect to these abstract definitions, while equalities and disequalities are treated as syntactic equality and disequality. Furthermore, we assume the expected properties of type union. For instance, if $p(num \cup char)$ is in a set, then $p(num)$ and $p(char)$ are both assumed to be in the set.

Definition 4.2 *Let P be a CLP program, and P_τ be the set $\{\mathcal{R}(Cl) \mid Cl \in P\}$. Its k-success set $SS^k[P_\tau]$ is defined inductively as*

$$SS^1[P_\tau] = \left\{ H \;\middle|\; H \leftarrow C \in ground(P_\tau), C \text{ is a true constraint} \right\}$$

$$SS^k[P_\tau] = \left\{ H \;\middle|\; \begin{array}{l} H \leftarrow B_1, \ldots, B_n, C \in ground(P_\tau), \\ C \text{ is a true constraint}, \\ \{B_1, \ldots, B_n\} \subseteq SS^{k-1}[P_\tau] \end{array} \right\}$$

$$SS[P_\tau] = \bigcup_{k=1}^{\infty} SS^k[P_\tau]$$

The notion of safety states, roughly, that any atoms in $SS[P]$ conforming to the given type declarations are represented in $SS[P_\tau]$.

Definition 4.3 *concretisation function*

Let $T \in Term_L^\tau$. Define the concretisation function for typed terms as $\gamma : Term_L^\tau \rightarrow 2^{Term_L}$ where:

$$\gamma(T) = \left\{ t \;\middle|\; \mathcal{R}(t) = T \right\}$$

The concretisation function is extended to atoms and sets of atoms. Let A be an atom. Then

$$\gamma(A) = \left\{ B \;\middle|\; \mathcal{R}(B) = A \right\}$$

Let $\mathcal{A}$ be a set of atoms. Then

$$\gamma(\mathcal{A}) = \bigcup \left\{ \gamma(A) \;\middle|\; A \in \mathcal{A} \right\}$$

The following lemma establishes a useful property about the reduction of well-typed expressions.

Lemma 4.4 *Let A and H be distinct well-typed expressions such that $A = H\theta$ (and assume $dom(\theta) \subseteq vars(H)$). Then $\exists\rho\ (\mathcal{R}(A) = \mathcal{R}(H)\rho)$.*

PROOF. Proof is by induction on the depth of H.

Base case: *depth* $= 1$. There are two cases.

1. H is a variable, say v. Then $\theta = \{v/A\}$, and $\rho = \{v/\mathcal{R}(A)\}$. $\mathcal{R}(v) = v$ so the property holds.

2. H is a constant, say c and so $\theta = \epsilon$ and $A = c$, and the property holds trivially.

Induction: Suppose $H = f(t_1, \ldots, t_n)$ is an expression of depth $k+1$, and $A = f(t_1\theta, \ldots, t_n\theta)$. Suppose first that f is a typed function symbol of type τ. Then $\mathcal{R}(A) = \mathcal{R}(H) = \tau$ since A and H are well-typed, and the property holds for this case with $\rho = \epsilon$.

Suppose f is an untyped function symbol. Then $\mathcal{R}(A) = f(\mathcal{R}(t_1\theta), \ldots, \mathcal{R}(t_n\theta))$. By the induction hypothesis $\mathcal{R}(t_j\theta) = \mathcal{R}(t_j)\rho_j$, where $\rho_j = \mathcal{R}(\theta)|_{vars(\mathcal{R}(t_j))}$, for $1 \leq j \leq n$. Hence $\mathcal{R}(A) = f(\mathcal{R}(t_1)\rho_1, \ldots, \mathcal{R}(t_n)\rho_n)$. Form the set ρ defined as

$$\rho = \bigcup_{j=1}^{n} \rho_j$$

ρ is a valid substitution since the same variable occurring in different substitutions ρ_i and ρ_k, say, are bound to the same term, by construction of ρ_i and ρ_k from θ. Hence

$f(\mathcal{R}(t_1)\rho_1, \ldots, \mathcal{R}(t_n)\rho_n) = f(\mathcal{R}(t_1), \ldots, \mathcal{R}(t_n))\rho$, which equals $\mathcal{R}(H)\rho$. Finally we note that $\rho = \mathcal{R}(\theta)|_{vars(\mathcal{R}(H))}$, by construction of ρ and the fact that

$$vars(\mathcal{R}(H)) = \bigcup_{j=1}^{n} vars(\mathcal{R}(t_j))$$

This completes the inductive case.

Therefore, the lemma holds for all expressions. □

We make the following assumptions about the safety of type reduction of constraints.

Assumption 4.5 *Let $p(x_1, \ldots, x_n)$ be a constraint atom where the predicate p/n has type $p(\tau_1, \ldots, \tau_n)$. Then $\mathcal{R}(p(t_1, \ldots, t_n)) = p(\tau_1, \ldots, \tau_n)$ for all true ground instances $p(t_1, \ldots, t_n)$.*

For every constraint $t_1 = t_2$ and $t_1 \neq t_2$, $\mathcal{R}(t_1) = \mathcal{R}(t_2)$ and $\mathcal{R}(t_1) \neq \mathcal{R}(t_2)$ respectively, where $=$ and $\neq$ between reduced terms are taken as free equality and disequality.

The latter assumption states that types are disjoint from each other and from non-typed terms.

Proposition 4.6 *Let P be a CLP program, RR be the set of rewrite rules giving types to all the constraint symbols (and possibly other symbols too), and P_τ be the program after applying type reduction to all its clauses. Let $SS[P]$ and $SS[P_\tau]$ be the success sets of P and P_τ respectively. If P is well-typed with respect to RR, then $SS[P] \subseteq \gamma(SS[P_\tau])$.*

PROOF. Let $SS^k[P]$ and $SS^k[P_\tau]$ be the sets of ground atoms as defined in Definitions 4.1 and 4.2. The proof is by induction on k.

Base Case: $k = 1$. Show that $SS^1[P] \subseteq \gamma(SS^1[P_\tau])$. Assume $p(\bar{t}) \in SS^1[P]$; show that $p(\bar{t}) \in \gamma(SS^1[P_\tau])$.

There exists $p(\bar{s}) \rightarrow \{C\} \in P$ where $\bar{t} = \bar{s}\theta$ and $C\theta$ is true. By Lemma 4.4 $\exists\rho$ such that $\mathcal{R}((p(\bar{s}) \rightarrow C)\theta) = \mathcal{R}(p(\bar{s}) \rightarrow C)\rho$. ρ grounds $\mathcal{R}(p(\bar{s}) \rightarrow C)$ and $\mathcal{R}(C)\rho = \mathcal{R}(C\theta)$ (by Lemma 4.4, which succeeds by Assumption 4.5). Hence $\mathcal{R}(p(\bar{s}))\rho \in SS^1[P_\tau]$, and by Lemma 4.4 $\mathcal{R}(p(\bar{s}))\rho = \mathcal{R}(p(\bar{s})\theta) = \mathcal{R}(p(\bar{t}))$. Hence $p(\bar{t}) \in \gamma(SS^1[P_\tau])$ (by definition of γ).

Induction: Assume that $SS^k[P] \subseteq \gamma(SS^k[P_\tau])$. Show that $SS^{k+1}[P] \subseteq \gamma(SS^{k+1}[P_\tau])$. Let $(p(\bar{s}) \rightarrow B_1, \ldots, B_m, \{C\})\theta$ be a ground instance of a clause in P, where $\{B_1\theta, \ldots, B_m\theta\} \subseteq SS^k[P]$, and $C\theta$ is true. Assume that $p(\bar{t}) \in SS^{k+1}[P]$ and that $p(\bar{t}) = p(\bar{s})\theta$; we show that $p(\bar{t}) \in \gamma(SS^{k+1}[P_\tau])$. By Lemma 4.4 there exists ρ such that $\mathcal{R}((p(\bar{s}) \rightarrow B_1, \ldots, B_m, \{C\})\theta) = \mathcal{R}(p(\bar{s}) \rightarrow B_1, \ldots, B_m, \{C\})\rho$.

By the induction hypothesis $\{B_1\theta, \ldots, B_m\theta\} \subseteq \gamma(SS^k[P_\tau])$. and $B_j\theta = \mathcal{R}(B_j)\rho$, $1 \leq j \leq m$, and hence $\{\mathcal{R}(B_1)\rho, \ldots \mathcal{R}(B_m)\rho\} \subseteq SS^k[P_\tau]$. By Assumption 4.5 $\mathcal{R}(C\theta)$ (which equals $\mathcal{R}(C)\rho$) succeeds. Therefore $\mathcal{R}(H)\rho \in SS^{k+1}[P_\tau]$. Hence $p(\bar{s})\theta$ (which equals $p(\bar{t})$) $\in \gamma(SS^{k+1}[P_\tau])$.

Hence the proposition holds for all $SS^k[P]$ and so $SS[P] \subseteq \gamma(SS[P_\tau])$. □

COROLLARY. Let P be a well-typed CLP program, and A a well-typed atom. Suppose $\mathcal{R}(A) \notin SS[P_\tau]$. Then $A \notin SS[P]$.

It is a simple matter to check whether a ground atom is well-typed, and the language compiler checks whether a program is well-typed. However membership of $SS[P_\tau]$ is undecidable, as is membership of $SS[P]$. Hence we apply the technique of regular approximation [13], [16] in order to find a decidable approximation of $SS[P_\tau]$. In other words, we will compute (a representation of) a superset of $SS[P_\tau]$, say T, for which membership is efficiently decidable. If $A \notin T$ then $A \notin SS[P_\tau]$ and hence given a well-typed atom A, the corollary provides a practical test for non-membership of well-typed atoms in $SS[P]$.

It is also decidable whether any definite goal has a solution in an RUL program. Hence, by testing the body of each clause in P_τ to see whether it has any solutions in the regular approximation of P_τ we have a test for "useless" clauses, that is, clauses that never yield any solutions (and are therefore presumably erroneous). This can be a useful debugging check, and also has applications in program specialisation [10].

5 Regular Approximation

The class of (canonical) RUL programs was defined by Yardeni and Shapiro [22]. The definitions in this section are quoted from Gallagher and de Waal [13].

Definition 5.1 *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 5.2 *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.*

Definition 5.3 *Let P be a definite program and P' an RUL program containing a special unary predicate $approx/1$. Let $M(P)$ denote the least Herbrand model of P. Then P' is a* **regular approximation** *of P if the least Herbrand model of P is contained in the set* $\left\{ A \;\middle|\; approx(A) \in M(P') \right\}$.

Example 2 *Let P be a program containing* `reverse(X,Y)` *where* `Y` *is the reverse of list* `X`. *Then a regular approximation of includes the following definition of* `approx`.

```
approx(reverse(X,Y)) :- t1(X), t1(Y).
t1([ ]).
t1([ X | Y ]) :- any(X), t1(Y).
```

Note that in using $approx/1$ we abuse notation by confusing predicate and function symbols but this could easily be corrected with more cumbersome notation. The use of the *approx* predicate allows us to restrict attention to RUL programs. Often we omit it and write a clause $approx(p(x_1,\ldots,x_n)) \leftarrow t_1(x_1),\ldots,t_n(x_n)$ as $p(x_1,\ldots,x_n) \leftarrow t_1(x_1),\ldots,t_n(x_n)$.

Some operations and relations on RUL programs are next defined.

Definition 5.4 *$success_R(t)$*

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

Definition 5.5 *intersection of regular 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 defined by an RUL program R', such that $success_{R'}(t_3) = success_R(t_1) \cap success_R(t_2)$.*

The use of regular approximations is a successful compromise between expressive power and computability. The intersection of regular unary predicates can be easily computed [22]. A tuple-distributive upper bound of regular unary predicates will be defined. First we introduce the concept of tuple-distributivity. We follow the presentation given in [18].

Definition 5.6 *Let T be a set of ground terms. Let $t \in T$ such that $f(s_1, \ldots, s_n)$ is a subterm of t. Let $u \in T$ be a term identical to t except that the subterm $f(s_1, \ldots, s_n)$ is replaced by $f(r_1, \ldots, r_n)$. We write $u = t[f(r_1, \ldots, r_n)/f(s_1, \ldots, s_n)]$.*

Then T is **tuple-distributive** *if for all such $t, u \in T$,*

$$\left\{ t[f(v_1, \ldots, v_n)/f(s_1, \ldots, s_n)] \;\middle|\; v_i = s_i \text{ or } v_i = r_i \right\} \subseteq T$$

If T is a set of terms then the smallest tuple-distributive set containing T is called the **tuple-distributive closure** *of T.*

Example 3 *If $T = \{f(a,b), f(c,d)\}$ then the tuple-distributive closure of T is*

$$\{f(a,b), f(a,d), f(c,b), f(c,d)\}$$

Definition 5.7 *upper bound of regular unary predicates*

Let t_1 and t_2 be unary predicates, defined in an RUL program R. Then the **tuple-distributive upper bound** *$t_1 \sqcup t_2$ is defined as a predicate t_3, such that $success_R(t_3) \supseteq success_R(t_1) \cap success_R(t_2)$ and $success_R(t_3)$ is tuple-distributive.*

Procedures for computing intersection and a tuple-distributive upper bound of predicates in RUL programs can easily be defined. Note that the tuple-distributive upper bound as defined in general includes the union of t_1 and t_2.

Definition 5.8 *upper bound of RUL programs*

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

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

Effective abstract interpretation procedures for computing a regular approximation of a program (using RUL programs or some equivalent notation) have been defined in [16] and [13]. Several other works have studied the problem. Our abstract semantic function is a mapping $\mathcal{T}_P$ from one RUL program to another. Termination is ensured by a "widening" operation [13] [16].

6 Approximation of Partly-Typed Programs

We now consider the problem of approximating a partly-typed logic program. In constraint logic programming languages some types are essential so that the constraint solvers can identify the appropriate solvers for numbers, booleans, tuples and so on. Our original motivation was to be able to apply regular approximation to constraint logic programs [20]. This is not possible without explicit handling of the typed constraint predicates and functions.

We consider two kinds of application.

1. Analysis of Prolog programs in which types are imposed on selected symbols.

2. Analysis of CLP programs in which the system types are taken into account and combined with untyped or user-typed symbols.

Although standard Prolog contains built-in predicates, one has the freedom to use functions such as arithmetic operators $+$, $-$, and so on with non-numeric arguments. However the programmer may wish to impose type restrictions on certain symbols, especially on arithmetic constants and functions and on list constructors [] and *cons*/2.

The approximation method consists of two stages:

1. A program is transformed to an abstract program, by replacing all typed symbols in the program by their types, using the type declarations as rewrite rules. We call this *type reduction*.

2. An established algorithm for regular approximation is applied to the transformed program. This gives a result which approximates the (well-typed) computations of the original program.

Example 4 *Consider the following example adapted from a program discussed in [1]. The predicate* `map_weight/2` *takes a list of words (here represented as lists of the atoms* `a, b` *and* `c`*) and computes a list in which each word is immediately followed by its weight.*

```
char_weight(a,1).
char_weight(b,2).
char_weight(c,3).

word_weight([], 0).
word_weight([X|Y],W) :- char_weight(X,S), word_weight(Y,V), W is S+V.

map_weight([],[]).
map_weight([X|Z],[X|[Y|W]]) :- word_weight(X,Y), map_weight(Z,W).
```

We use the following type declarations.

```
[] --> list(bottom)
[X | list(Y)] --> list(union(X,Y))
0 --> num
1 --> num
2 --> num
. . .
num + num -->  num
```

Note that, unlike the example in [1], we assume no types for the symbols `a, b` *and* `c`*. We could easily define types for these if desired. Type reduction yields the following program. We assume that the builtin predicate* `X is Y` *is approximated by the fact* `num is num`*.*

```
char_weight(a,num):-true.
char_weight(b,num):-true.
```

```
char_weight(c,num):-true.
word_weight(list(bottom),num):-true.
word_weight(list(Z1),X3):-
      union(X1,X2,Z1),
      char_weight(X1,num),word_weight(list(X2),num),X3 is num.
map_weight(list(bottom),list(bottom)):-true.
map_weight(list(Z2),list(Z4)):-
      union(X1,Z3,Z4),union(X3,X4,Z3),union(X1,X2,Z2),
      word_weight(X1,X3),map_weight(list(X2),list(X4)).
```

6.1 Regular Approximation of Type-Reduced Programs

We next consider how to compute an approximation of $SS[P_\tau]$ for a type-reduced program P_τ. The regular approximation procedure, described in [13] is a bottom-up fixpoint computation. The initial approximation is empty. In each iteration the body of every program clause is solved with respect to the current approximation; the next approximation is the tuple-distributive upper bound of the inferred approximation of the clause heads. The procedure terminates when an iteration does not increase the approximation. Various optimisations of the basic procedure are described in [13] and it has been shown to scale up well to larger programs.

6.1.1 Tuple-Distributive Union Types

If the type union symbol is not present in P_τ then regular approximation of P_τ can be computed directly. If the type union symbol is present then we use an assumption called the Tuple-Distributive Union Type assumption during regular approximation.

Definition 6.1 *TDU*

Let A be an atom and suppose that $A[t]_p$ means that t occurs at a position p in A. The Tuple-Distributive Union Type assumption (TDU) states that

$$A[\tau_1 \cup \tau_2]_p \leftrightarrow A[\tau_1]_p \wedge A[\tau_2]_p$$

The TDU induces a safe approximation. It just implies that we lose the distinction between a union type and a set of individual types at the same argument position. For example, if a predicate $p/1$ can succeed for both $p(list(num))$ and $p(list(char))$, we will infer that its type is $p(list(num \cup char))$ and vice versa. This is clearly an over-approximation but does not seem to be a serious loss of information for typical programs. The TDU hypothesis appears to be related to assumption in other program analysis frameworks [8].

Using the TDU, we can reduce an occurrence of $\tau_1 \cup \tau_2$ to occurrences of τ_1 and τ_2. We can then interpret type union using the tuple-distributive upper bound on RUL predicates.

- Firstly, suppose P_τ contains a clause $H \leftarrow B1, \ldots, B_j[\tau_1 \cup \tau_2], \ldots, B_n$. By the TDU this is equivalent to $H \leftarrow B1, \ldots, B_j[\tau_1], B_j[\tau_2], \ldots, B_n$
- Secondly, suppose P_τ contains a clause $H[\tau_1 \cup \tau_2] \leftarrow B1, \ldots, B_n$ By the TDU this is equivalent to the two clauses $H[\tau_1] \leftarrow B1, \ldots, B_n$ and $H[\tau_2] \leftarrow B1, \ldots, B_n$.

In our implementation, type unions in clause bodies are handled by a transformation as described in the first case above. The length of the clause body is thus increased but the number of clauses remains the same. Type unions in the head could be handled by the transformation described in the second case, but it would be rather inefficient since the number of clauses can multiply very quickly. Instead, the regular approximation procedure is modified slightly to achieve the same result more efficiently. Let $H[\tau_1 \cup \tau_2] \leftarrow B$ (where $\cup$ has been eliminated from B) be a clause. In the bottom-up interpretation procedure the body B is first solved. This yields regular descriptions of the terms τ_1 and τ_2. The tuple-distributive upper bound of these descriptions is computed and used to infer the approximation for H.

Example 5 *Consider the type-reduced program in Example 4. Regular approximation is applied to that program and the following approximation for* `map_weight/2` *is obtained.*

```
map_weight(X1,X2) :-t53(X1),t54(X2).

t53(list(X1)) :-t47(X1).          t54(list(X1)) :-t51(X1).
t47(list(X1)) :-t33(X1).          t51(list(X1)) :-t33(X1).
t47(bottom) :-true.               t51(num) :- true.
t33(c) :-true.                    t51(bottom) :- true.
t33(a) :-true.
t33(b) :-true.
t33(bottom) :-true.
```

Applying the tuple-distributive type union assumption, and eliminating `bottom`, the first argument thus has type `list(list({a,b,c}))`. The second has type `list(num` $\cup$ `list({a,b,c}))`.

7 Experimental Results

7.1 Analysis of Precision of Regular Approximation

Figure 1 shows the results for a selection of test programs. Most of these are standard test programs in the literature. The figures in the table show the results for bottom-up, and top-down analysis. Top-down analysis is simulated using a version of query-answer transformations [13], [11]. An uninstantiated top goal is given as the query in the top-down analysis. The implementation is in SICStus Prolog running on a SPARC-10.

For most of these programs only lists and numbers are typed in our analysis. Many of the programs contain other untyped symbols.

In order to get a rough measure of precision, we count the percentage of predicate arguments for which the analysis determines the type or principal functor (the columns marked P in the figure). This measure is actually rather limited, especially for bottom-up analysis where most of the interesting results are below the predicate argument level. However it serves to show the increase in precision gained by top-down analysis.

The timings show two things. Firstly, the bottom-up analysis performs well on larger examples and suggests that the analysis time is roughly proportional to the size of the program. The difference between bottom-up and top-down analysis is more variable, with the time for top-down being roughly 2 to 9 times more than for bottom-up.

		Bottom-Up		Top-Down	
Program	Number of Clauses	Time (secs)	Precision	Time (secs)	Precision
append	2	0.01	33	0.04	33
balance	10	0.08	75	0.59	100
disj_r	71	0.8	46	6.97	88
gabriel	45	0.42	47	1.99	81
kalah_r	88	0.8	52	7.8	89
map_weight	8	0.06	100	0.13	100
peep	227	1.47	55	6.35	68
pg	18	0.28	48	1.38	90
plan	29	0.19	66	1.62	75
press	162	1.34	52	12.49	71
qsort	6	0.12	56	0.49	89
queens	9	0.08	64	0.44	82
rotate	6	0.07	49	0.1	100

Figure 1: Timings and Precision for Sample Programs

Preliminary conclusions of the evaluation of precision show that the fast bottom-up analysis can capture a reasonable proportion of the results gained by the slower top-down analysis. Bottom-up may therefore be considered as a useful analysis, since query-dependent results can be deduced very quickly from bottom-up analysis too. Note that the results shown are only success types; query types with respect to a top goal and left-to-right computation rule can efficiently be computed from success types.

8 Related work

8.1 Type Systems and Success "Types" in Logic Programming

Our work combines aspects of two different notions of types in logic programming. One is the classical approach where types are used to give the intended interpretation of symbols (prescriptive typing). Badly-typed expressions are simply meaningless. On the other hand, a notion of type in logic programming has grown up, based on the success set of a program (descriptive typing) . In other words, the type associated with a predicate is a superset of the success set for that predicate. The two notions are quite distinct, though for definite logic programs there is quite a close practical correspondence between them. However, for a wider class of programs, the connection between "type" and "success set" breaks down. For this reason we prefer to reserve the term "type" for the classical notion. We combine prescriptive typing and descriptive typing using abstract compilation for prescriptive typing, a bottom-up or top-down abstract interpretation for descriptive typing.

In addition to the distinction between classical types and success set types, there is the difference between type inference and type checking. Type inference is the derivation of types for symbols for which no types are declared. Type checking is concerned with whether

declared (and inferred) types are used consistently.

Polymorphic type systems have been broadly studied in Functional Programming (see Cardelli and Wegner [5] for a survey). There are several proposals for polymorphic types for logic programming. Types for logic programs were introduced by Bruynooghe [2]. One of the first proposals for a polymorphic type system is the one of Mycroft and O'Keefe [19]. In their proposal the programmer has to declare types for predicates and functions, and a static type checker checks that the declarations are respected. Dietrich and Hagl [12] propose a polymorphic type system for Prolog based on the system of Mycroft and O'Keefe. They extend that system to deal with subtypes, when information about the possible dataflow in a clause is available. In [14] Hanus shows that polymorphic types allows the application of higher order programming techniques in Prolog, following a declarative approach in which the programmer has to declare all types of functions and predicates he wants to use in the program.

8.2 Type Inference Based on Success Types

Mishra [18] define a type inference method for deriving a tuple-distributive regular description of a superset of a program's success set. The type of a predicate describes (at least) all terms for which the predicate may succeed. The inferred types are symbolic descriptions of terms by means of ground regular trees. Yardeni and Shapiro [22] define a class of types called regular types, and a subclass of logic programs, called regular unary logic programs, and show the equivalence between them. They represent regular types by Deterministic Finite Automata (DFA) and use tuple distributive closure for types as defined by Mishra [18]. We adopted their formalism for describing sets of terms. In [23], Zobel presents an approach similar to the one in [18]. A syntactic type inference based on a specialisation of the unification algorithm is presented. No type declarations are required. The clauses are explored only once, to ensure termination, so type information is rather imprecise in the case of recursive clauses. Dart and Zobel [9] define a broader class of regular types, since they do not assume tuple distributivity. They give a lemma which describes a necessary condition for types to be regular. It is not a sufficient condition since some non-regular types satisfy that lemma. How to make the lemma more specific is left as an open problem.

Bruynooghe and Janssens [4], Gallagher and de Waal [13], Van Hentenryck, Cortesi and Le Charlier [16] and Heintze and Jaffar [15] all present algorithms based on computing decidable descriptions of the success set of a program. Various different representations were used in these works, including type graphs, regular unary logic programs, and set expressions. In the context of functional languages, Jones [17] obtained regular descriptions of the arguments of functions using regular sets of trees. These methods differ in their approach to widening. Bruynooghe and Janssens [4] restrict the number of occurrences of a functional symbol on the paths of types graphs, while the widening operator used in [16] is based on the notion of a "topological clash" between successive approximations. A "widening" operation is used in [13] based on the function symbols appearing in each regular procedure. This ensures termination without losing too much precision. Essentially, all these systems compute a superset of a program's success set and do not deal with polymorphic types. The systems implemented in [16] and [13] have been shown to perform well and to scale up well to larger programs.

Xu and Warren [21] described a type inference system which, like ours, transforms a program by replacing typed symbols by their types. The type inference is then performed by

a theorem prover.

Barbuti and Giacobazzi [1] introduce a polymorphic type inference system based on a bottom-up abstract interpretation. Types are declared for all the function symbols of a program, and an abstract interpretation procedure infers success types for the predicates. They first give a version handling polymorphic homogeneous type declarations, and then extend it for heterogeneous types with a type union operator. One important difference is that they ensure termination using the depth-k cutting of terms. We use the shortening operator described in [13] to create recursive descriptions and ensure termination. The other difference from our work is that we mix regular approximation of terms containing untyped function symbols with type inference using declared types. No experimental results are reported in [1].

Codish and Demoen [6] also developed a polymorphic type inference algorithm based on abstract interpretation. As in [1], they assume that every function symbol has a declared type. The algorithm employs "abstract compilation", in other words, the concrete program is first transformed into an abstract program in which the abstract operations are embedded. The concrete meaning of the abstract program is then computed. Our method uses abstract compilation in the first stage, but then uses an abstract interpretation (regular approximation) of the transformed program. Termination of their analysis is ensured using depth-k cutting of terms. The precision and conciseness of their results are impressive since accurate variable sharing between terms is captured using the "PROP" domain. The preliminary results in [6] suggest that computation time diverges rapidly for larger programs.

In [3] Bruynooghe and Boulanger show informally that frameworks developed for abstract interpretation of logic programs can be carried over to constraint logic programming.

Query-answer transformations, which we used to simulate top-down flow analysis in a bottom-up framework, are described in [11], and their use is discussed in [13].

9 Conclusions and Future Research

The initial motivation for this work was to extend regular approximation algorithms for Prolog to constraint logic programs. In order to do this we found that types and regular descriptions had to be mixed. In the course of the work it became clear that partly-typed programs are useful in other contexts, especially in untyped languages in which the user wishes to impose some type restrictions.

We have presented an abstract interpretation scheme based on regular approximation, exploiting type information given, to analyse logic programs which are partly typed. The method consists of a transformation followed by regular approximation. Parametric polymorphism with union types is handled. This method can be used to get descriptions of the success sets of constraint logic programs, which are typically partly typed, or untyped logic programs in which some symbols have types imposed by the user. At one extreme, if no types are declared, then the algorithm is just regular approximation of the success set of the program. At the other extreme, if all function symbols have declared types, then it just infers types for the predicates. Even in logic programming systems in which predicate types are given (such as Gödel), the algorithm might be used to infer more precise types for restricted uses of general predicates.

Experimental results show that the algorithm performs quite well on larger programs.

One application which we have investigated and intend to continue is a practical procedure for performing debugging on programs by detecting clauses that are either badly-typed, or useless (that is, never succeed), or both.

Applications of type inference include deriving information for compiler optimisations, debugging, compile time pipe-line optimisations, other cases where two or more goals are executed in conjunction, and other specialisations.

Several directions for further research can be mentioned. One is to increase the precision of regular approximation by incorporating sharing information. The use of the "PROP" domain used in [6] appears a promising approach. A more precise handling of union types, avoiding the tuple-distributive union type assumption, also seems feasible. It is possible to improve the shortening operator in our regular approximation algorithm to ensure termination but get better precision.

Acknowledgements

We would like to thank André de Waal, Manolis Marakakis, Estrella Pulido and Bern Martens for discussions throughout the course of this work.

References

[1] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19:281–313, 1992.

[2] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference*, pages 129–133, 1982.

[3] M. Bruynooghe and D. Boulanger. *Abstract Interpretation for (Constraint) Logic Programming*. Technical Report Report CW 183, Department of Computing Science, K.U. Leuven, November 1993.

[4] 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.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[6] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proceedings of the First Symposium on Static Analysis*, pages 281–297, Springer-Verlag, September 1994.

[7] 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.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 269–282, ACM Press, New York, U.S.A., 1979.

[9] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187, MIT Press, 1992.

[10] D.A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12), Nancy*, 1994.

[11] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[12] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proceedings of the 2nd European Symposium on Programming, Berlin*, pages 79–93, Springer-Verlag, 1988.

[13] J. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, MIT Press, 1994.

[14] M. Hanus. Polymorphic higher order programming in Prolog. In *Proceedings of the Sixth International Conference and Symposium on Logic Programming, Cambridge*, pages 382–397, MIT Press, 1989.

[15] 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.

[16] 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.

[17] 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.

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

[19] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[20] H. Sağlam and J. Gallagher. *Approximating Logic Programs Using Types and Regular Descriptions.* Technical Report CSTR-94-19, Department of Computer Science, University of Bristol, December 1994.

[21] Y. Xu and D. S. Warren. A type inference system for Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming, Seatle*, pages 604–619, MIT Press, 1988.

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

[23] 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.