

---

## A TYPE SYSTEM FOR LOGIC PROGRAMS\*

EYAL YARDENI AND EHUD SHAPIRO

---

- ▷ A theory for a type system for logic programs is developed which addresses the question of well-typing, type inference, and compile-time and run-time type checking. A type is a recursively enumerable set of ground atoms, which is tuple-distributive. The association of a type to a program is intended to mean that only ground atoms that are elements of the type may be derived from the program. A declarative definition of well-typed programs is formulated, based on an intuitive approach related to the fixpoint semantics of logic programs. Whether a program is well typed is undecidable in general. We define a restricted class of types, called regular types, for which type checking is decidable. Regular unary logic programs are proposed as a specification language for regular types. An algorithm for type-checking a logic program with respect to a regular type definition is described, and its complexity is analyzed. Finally, the practicality of the type system is discussed, and some examples are shown. The type system has been implemented in FCP for FCP and is incorporated in the Logix system.
- 

### 1. INTRODUCTION

Type checking within the framework of logic programming is useful for several reasons:

Type declarations and compile-time type checking provide a measure of confidence in the correctness of a program.

Typing a program makes it clearer and more readable.

---

\*This paper is a major revision of [23].  
Address correspondence to Eyal Yardeni, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel.  
Received November 1987; accepted December 1989.

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1991  
655 Avenue of the Americas, New York, NY 10010

0743-1066/91/\$3.50

Errors in logic programs tend to fall into several broad categories. One of these is bad interfacing between predicates. This can be caused by incorrect ordering of arguments in a call, by an incorrect type of an argument being passed, or by misspelling variable names. Some of these errors can be detected with the aid of a type checker, and their detection is especially beneficial in a system which incorporates modules, such as Logix [18].

This paper formulates a type theory for pure logic programs and develops a type-checking algorithm. Three aspects of the type system are introduced: type declaration, type inference, and well-typing.

This theoretical model is a suitable basis for type systems for concrete logic-programming languages. For this purpose, a new interpretation of types is introduced, and a novel method for performing type checking is developed.

The paper defines the class of regular types [14] and a subclass of logic programs, called regular unary logic programs, and shows the equivalence between them. The syntax of regular types is defined by a BNF notation, and any regular type can be implemented by deterministic finite automata (DFA). The incarnation of regular types as DFA is useful for showing inclusion, union, and equivalence of regular types.

By type inference we mean the relation of a type to the semantics of a logic program. A type may be associated with any logic program using a fixpoint of a function. This function is an abstraction of the usual semantic function of logic programs, and corresponds to the intuition that every atom in the type is inferable from the program.

In general, the question of whether a program is well typed according to our definition is undecidable. However, any particular computation can be checked not to violate a particular type specification by a run-time check. This check could be expensive. For regular types, well-typing is decidable. A compile-time type-checking algorithm is developed for regular types, and its complexity is analyzed.

The significance of this work is in the formalization of a type theory for logic programs that gives a single formal framework which includes a theory for type declaration and checking based on a sound mathematical foundation.

Bruynooghe [3] suggests adding types and modes (input or output) to a logic program. In this system each argument has a mode annotation, and using this knowledge he develops an algorithm for type checking. The algorithm is not described within any theoretical framework, but is only shown to work on several programs.

Mycroft and O'Keefe [16] adopt the outlook of Milner [13] that well-typed programs cannot go wrong. They treat PROLOG as a procedural or an applicative language—each procedure and each function call conforms to the type declaration of that object. Each construct in the language is associated with a type, and using that information, well-typing is defined. They also define a resolvent to be well typed if each atom in the resolvent is well typed. Afterwards they prove that if the program and the goal are well typed, then the following hold:

- (1) The resolution at each step of the computation is well-typed.
- (2) The variables of the top-level goal can only be instantiated to values having types as dictated by the type declaration.

The semantics of their type is not defined, type equivalence is checked only syntactically, and subtyping is not dealt with. Dietrich [5] enhanced Mycroft and O'Keefe's type checker to deal with polymorphic subtypes by adding modes to the program.

Quite a few papers have been published on type inference for variants of logic programming. Mishra [14] made the first attempt at type inference. Following him, Zobel [25] did polymorphic type inference for PROLOG with the possibility of combining it with type declaration. Kluzniak [10] developed a type inference system for ground PROLOG. Fruhwirth [6] uses program transformation and partial evaluation to infer the type of the program, and Kanamori, Horiuchi, Kawamura, Bruynooghe, and Janssens [7,8,2] use abstract interpretation techniques to do type inference. These papers define algorithms that identify clauses, which cannot be used in a successful computation. On the other hand, the type-checking papers are concerned with finding inconsistent clauses, i.e., either atoms in the clauses that do not conform to the type declaration or usage of the same variable in disjoint types. Xu and Warren [22] also implemented a type inference system using depth abstraction and type definitions. They combined it with type declarations that restrict the domain of predicates. A program is well typed according to their definition if the type inference procedure succeeds and produces a type that is a subset of the declared type. Our paper is oriented towards type checking; hence well-typing and the theory that it involves are developed. Our work has similarity to that of Xu and Warren in that the semantics of the program is influenced by the type declarations.

The paper is organized as follows. Section 2 introduces regular types, RUL programs, the equivalence between the two, and some useful properties of them. Section 3 investigates an abstraction of the well-known  $T_p$  operator. The domain of the abstract operator is the *tuple-distributive* sets of atoms from the Herbrand base. Section 4 defines the notion of well-typing. Section 5 describes how to declare a type and gives some examples. Section 6 develops a type-checking algorithm and considers its complexity. Section 7 describes the implementation of the type system and our experience with it. Section 8 concludes the paper.

## 2. TYPES

We define the notion of types and then introduce a subset of types that can be represented by DFA. Later we define regular unary predicate logic programs (RUL programs). RUL programs are unary logic programs that obey certain syntactic rules. We use RUL programs to simulate DFA that represent regular sets of terms. We have chosen RUL programs instead of DFA in order to achieve a more readable representation of types. In addition, RUL programs are logic programs. So run-time type-checking can be easily achieved by incorporating in the program RUL programs that define types, and restricting each argument of the head to its type declaration. Finally we present a BNF syntax of typed programs.

### 2.1. General Overview

*Definition.* Let  $S$  be a set of first-order formulas, and  $L$  a first-order language. The *signature* of  $S$ ,  $sig(S)$ , is the minimal set containing all predicates, function

symbols and constants that appear in  $S$ . Similarly we define the *signature* of  $L$ ,  $\text{sig}(L)$ , to be the signature of all formulas that can be constructed in  $L$ .

NOTE. To facilitate metaprogramming, we assume one set of symbols for both functions and predicates.

Let  $L$  be a first-order language. The *Herbrand universe*  $H_L$  and the *Herbrand base*  $B_L$  are defined as usual. A logic program  $P$  defines a signature. We use the symbol  $B_P$  as a synonym for  $B_{\text{sig}(P)}$  to denote the Herbrand base of  $P$ .

*Notation.*  $A \ll_L B$  means that  $A$  is a ground instance of  $B$  over  $L$ . When  $L$  is understood, we omit the subscript.

*Definition.* Let  $B_P$  be the Herbrand base of a program  $P$ . Define the mapping  $T_P: 2^{B_P} \rightarrow 2^{B_P}$  as follows: Let  $I$  be a Herbrand interpretation. Then

$$T_P(I) = \{A \in B_P \mid A \leftarrow B_1, \dots, B_n \leftarrow C \in P \text{ and } B_1, \dots, B_n \in I\}.$$

Van Emden and Kowalski [21] prove that the intersection of all Herbrand models for  $P$  is equal to  $T_P \uparrow \omega$ , where  $\omega$  is the first infinite ordinal, and call it the *meaning* of the program. We denote by  $\llbracket P \rrbracket$  the meaning of a program  $P$ .

## 2.2. Tuple Distributivity

The notion of tuple-distributivity is due to Mishra [14].

*Definition.* With each term  $t$  we associate a labeled tree, which we refer to as the *associated tree* of  $t$ . The edges and the leaves of the tree are labeled according to the following construction rules:

- (1) If  $t$  is a constant or a variable symbol, then make a leaf and label it with  $t$ .
- (2) If  $t = f(t_1, t_2, \dots, t_n)$ ,  $f$  is of arity  $n$ , then:
  - (a) Make a new node associated with  $t$ .
  - (b) For all  $i \in [1..n]$  construct recursively the subtree associated with  $t_i$ , and draw an edge, labeled  $f(n, i)$ , from the new node to the root of the subtree.

An example is given in Figure 1.

*Definition.* Let  $t$  be a term. A *path* in  $t$  is the sequence of labels between the root and a leaf of the tree associated with  $t$ . The set of all paths of the associated tree of  $t$  is denoted by  $\text{paths}(t)$ . A *node* in  $t$  is the node of the associated tree of  $t$ . The set of all paths between a node and the leaves in the associated tree of  $t$  is called *paths of the node* of  $t$ . A path is *ground* if it does not end in a node labeled by a variable.

*Example.*  $\text{paths}(f(g(h(a), b), c)) = \{f(2, 1)g(2, 1)h(1, 1)a, f(2, 1)g(2, 2)b, f(2, 2)c\}$ .

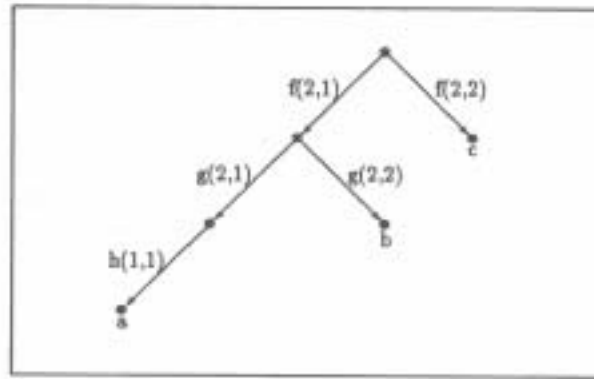


FIGURE 1. The associated tree of  $f(g(h(a), b), c)$ .

*Definition.* Let  $S$  be a set of ground terms.

Define  $paths(S) = \bigcup_{t \in S} paths(t)$ .

The *tuple-distributive closure* of  $S$  is

$$\alpha(S) = \{t \mid t \text{ is a term and } paths(t) \subseteq paths(S)\}.$$

$S$  is *tuple-distributive* if  $\alpha(S) = S$ .

Intuitively, the tuple-distributive closure of a set of terms is the set of all terms constructed recursively by permuting each argument position among all terms that have the same functor-arity combination.

*Example.* If  $S = \{f(a, b), f(c, d)\}$ , then  $paths(S) = \{f(2, 1)a, f(2, 2), b, f(2, 1)c, f(2, 2)d\}$  and  $\alpha(S) = \{f(a, b), f(a, d), f(c, b), f(c, d)\}$ .

**REMARK.**  $\alpha$  is idempotent, i.e.,  $\alpha(\alpha(S)) = \alpha(S)$  for every set of ground terms  $S$ .

*Definition.* Let  $p/n$  be a predicate in a program  $P$ . Then

$$\llbracket P \rrbracket_{p/n} = \{p(t_1, \dots, t_n) \mid p(t_1, \dots, t_n) \in \llbracket P \rrbracket\}.$$

We extend the notions of paths and tuple-distributivity to include atoms.

*Claim.* Let  $P$  be a program with different predicates  $p_1, \dots, p_n$ . Then  $\alpha(\llbracket P \rrbracket) = \alpha(\llbracket P \rrbracket_{p_1}) \cup \dots \cup \alpha(\llbracket P \rrbracket_{p_n})$ .

*Definition.* A *type* is a recursively enumerable (r.e.) tuple-distributive set of ground atoms with a finite signature.

We require that types be r.e. so that type restriction can be incorporated in the program.

### 2.3. Regular Types and Their Properties

*Definition.* Let  $S$  be a type or a program. Define

$$\Sigma_S = \{f(n, t) \mid f/n \in \text{sig}(S), i \in \{1, \dots, n\}\} \cup \{c \mid c \in \text{sig}(S) \text{ is a constant}\}.$$

*Definition.* A set of ground terms  $S$  with a finite signature is *regular* iff there exists a regular language  $\mathcal{L} \subseteq \Sigma_S^*$  s.t. for every term  $t$ ,  $t \in S$  iff  $\text{paths}(t) \subseteq \mathcal{L}$ . A logic program  $P$  is *regular* if  $\llbracket P \rrbracket$  is regular.

This definition of regular sets is equivalent to the one based on deterministic root-to-frontier tree automata [15, 20]. Note that every regular set of terms is tuple-distributive.

*Example.* The set of terms  $S = \{t(X, X) \mid X = s^n(0), n < \omega\}$  is not regular. By contradiction assume that  $S$  is a regular set of terms. Since  $t(0, 0), t(s(0), s(0)) \in S$ , then by tuple distributivity  $t(0, s(0)) \in S$ . Contradiction.  $\square$

*Claim.* The intersection of regular sets of terms is regular.

*PROOF.* Trivial, by the fact that the intersection of regular languages is regular.  $\square$

*Example.* The union of regular sets is not necessarily regular. Let

$$S1 = f(a, b),$$

$$S2 = f(c, d);$$

then

$$S3 = S1 \cup S2$$

is not regular. To see that, note that

$$\text{paths}(f(a, b)) = \{f(2, 1)a, f(2, 2)b\}.$$

$$\text{paths}(f(c, d)) = \{f(2, 1)c, f(2, 2)d\}.$$

Assume that  $S3$  is regular. Then there exists a regular language  $\mathcal{L}$  s.t.  $t \in S3$  iff  $\text{paths}(t) \subseteq \mathcal{L}$ . Clearly  $\mathcal{L}$  contains  $\text{paths}(f(a, b)) \cup \text{paths}(f(c, d))$ . But then  $f(a, d)$  is also in  $S3$ . Contradiction.  $\square$

This example shows also the difference between a union of deterministic root-to-frontier tree automata and a union of automata over strings.

If  $S$  is a regular set of terms, then we know that there exists a regular language  $\mathcal{L}$  s.t.  $t \in S$  iff  $\text{paths}(t) \subseteq \mathcal{L}$ . The following lemma shows that  $\text{paths}(S)$  is an example of a regular language fitting this definition.

*Lemma 1.* For every regular set of terms  $S$ ,  $\text{paths}(S)$  is regular.

*PROOF.* See Lemma A.4 in the appendix.  $\square$

By Lemma 1, it is clear that  $paths(S)$  is the minimal regular language  $L$  such that  $t \in S$  iff  $paths(t) \subseteq \mathcal{L}$ .

*Claim.* Let  $S_1$  and  $S_2$  be regular sets of terms. Then

$$S_1 \subseteq S_2 \text{ iff } paths(S_1) \subseteq paths(S_2).$$

PROOF. Trivial from the definition.  $\square$

#### 2.4. RUL Programs

*Definition.* Two terms are *top-level unifiable* if at least one of them is a variable or they have the same principal function symbol.

*Definition.* A *regular unary logic (RUL) program*  $P$  is a logic program satisfying the following syntactic rules:

- (1) Every predicate in  $P$  is unary.
- (2) No two head arguments of clauses of the same predicate are top-level unifiable.
- (3) Every body goal of every clause in  $P$  is of the form  $p(X)$ , where  $p$  is a predicate name and  $X$  is a variable.
- (4) Every variable in a clause occurs exactly once in its head and once in its body.

Note that the arguments of facts (clauses with empty bodies) are ground terms [implied by rule (4)].

*Example.* The following is a RUL program:

```

procedure(merge_all(Xs, Ys)) ←
  list_of_nat_lists(Xs),
  list_of_nat_lists(Ys),
  list_of_nat_lists([]),
  list_of_nat_lists([X|Xs]) ←
    nat_list(X),
    list_of_nat_lists(Xs),
  nat_list([]),
  nat_list([X|Xs]) ←
    natural(X),
    nat_list(Xs).

```

*Definition.* Let  $p$  be a unary predicate and  $A$  a set of atoms. Define  $A/p$  to be the set  $\{t \mid p(t) \in A\}$ .

*Theorem 1.*

- (1) RUL programs are regular.
- (2) For every regular set of terms  $S$  there exists a RUL program  $P$  with a predicate  $p$  such that  $S = \llbracket P \rrbracket / p$ .

PROOF. The proof is a result of Lemmata A.1, A.2, and A.3 in the appendix.  $\square$

Using an example, we show that RUL programs can be easily represented by BNF derivation rules:

*Example (Isomorphism of binary trees with labeled nodes).* The type definition by a RUL program:

```
natural(0).
natural(s(X)) ←
  natural(X).

binary_tree(void).
binary_tree(tree(X, Y, Z)) ←
  natural(X),
  binary_tree(Y),
  binary_tree(Z).

procedure(isotree(Y, Z)) ←
  binary_tree(Y),
  binary_tree(Z).
```

The BNF derivation rules for the same type:

```
Natural ::= 0 ; s(Natural).
Binary_tree ::= void ; tree(Natural, Binary_tree, Binary_tree).
Procedure ::= isotree(Binary_tree, Binary_tree).
```

To summarize, if types are defined by RUL programs, then:

- (1) Type intersection can be done by intersecting the corresponding automata and building a new RUL program from the intersected automaton.
- (2) The inclusion of one type in another can be decided by checking the inclusion of the minimal regular languages that represent the types. Algorithms for inclusion, intersection, and union of regular languages are well known [1].

These results are used in Section 6 on type checking.

### 3. TYPE INFERENCE

This section is based on the semantics of logic programs described by van Emden and Kowalski [21] and by Lloyd [12].

A formal definition of well-typing is given below. Here we would like to provide some intuition for our approach. We assume that  $L$  is a first-order language. Henceforth, we also assume that all programs are written in that language.

The assumption of a fixed global vocabulary precludes composition in its most general form. It seems that compositionality and parametrized (polymorphic) types are closely related.<sup>1</sup>

We define a rule of type-relative inference by a program  $P$  with respect to a type  $S$ .

<sup>1</sup>An extension of our approach to parametric types is described in a subsequent paper [24].

*Definition.*

We say that  $A$  is *inferred by  $P$  relative to  $S$*  iff  $A \in \alpha(T_P(S))$ .

A clause  $C$  is *useless relative to  $S$*  if  $T_{\{C\}}^\alpha(S)$  is the empty set.

In other words, a ground atom  $A \in B_P$  is inferred by  $P$  relative to  $S$  if there exists a clause  $C$  in  $P$  and atoms  $B_1, B_2, \dots, B_n \in S$  such that  $A \leftarrow B_1, \dots, B_n$  is an instance of  $C$ . In addition, any atom in the tuple-distributive closure of atoms inferred by this rule is also defined to be inferred by  $P$  relative to  $S$ .

We would like the type declaration  $S$  to be an approximation to the meaning of  $P$ . So we define that a program in our system is *well-typed by  $S$*  if it infers all atoms in  $S$  relative to  $S$  and only these atoms. In addition, we require that all clauses in the program be useful, i.e., each clause can infer at least one atom in  $B_L$  relative to  $S$ .

This leads to the idea that the well-typing notion of a program can be stated as a fixpoint of an operator which is related to the program's semantics (to be defined below).

The relation between our work and abstract interpretation is described in the next subsections.

*3.1. Abstract Interpretation*

A program denotes computations in some universe of objects. Abstract interpretation of programs uses that denotation to describe computations in another universe of abstract objects, so that the computation in the abstract domain is effective and yields some information about the standard denotation [4].

Our notion of well-typing is defined below in terms of an abstract interpretation. We show that types can be viewed as an abstraction of meanings. We show interesting properties that relate the concrete domain to the abstract domain and the type system. We describe an interpreter that operates in the abstract domain and is an abstraction of the well-known interpreter for logic programs.

We define an abstract interpretation of logic programs as follows: The concrete domain is  $2^{B_L}$ , and the abstract domain is the set of all tuple distributive sets in  $2^{B_L}$ .

The abstraction function,  $\alpha(S) = \{t \mid t \text{ is a term and } \text{paths}(t) \subseteq \text{paths}(S)\}$ , was defined in the previous section. The concretization function  $\gamma$  is the identity function.

We define a new operator, which is an approximation to  $T_P$  and operates on the abstract domain.

*Definition.* Let  $P$  be a program and  $X$  a type. Define  $T_P^\alpha$ , the abstract function of  $T_P$ , to be

$$T_P^\alpha(X) = \alpha(T_P(X)).$$

From the definition of *paths* we see that every tuple-distributive set can be represented uniquely by its paths. Then the above function is equivalent to

$$T_P^\alpha(Y) = \{X \mid X \in \text{paths}(A), \text{paths}(A) \subseteq \text{paths}(B_P), A \leftarrow B_1, B_2, \dots, B_n \leftarrow C, \\ C \in P, \text{ and } \text{paths}(\{B_1, \dots, B_n\}) \subseteq Y \}$$

in the sense that they represent the same set, i.e.,

$$\text{paths}(T_P^n(X)) = T_P^n(\text{paths}(X)).$$

*Definition.* Let  $C$  be a complete lattice, ordered by the inclusion relation.

Let  $X \subseteq C$ .  $X$  is *directed* if every finite subset of  $X$  has an upper bound in  $X$ .

Let  $T: C \rightarrow C$  be a function.  $T$  is *continuous* if  $T(\text{lub}(X)) = \text{lub}(\{T(I) \mid I \in X\})$  for every directed subset  $X$  of  $C$ .

*Lemma 2.*  $T_P^n$  is continuous over the set of tuple-distributive elements in  $2^{B_C}$ .

*PROOF.* Let  $X$  be a directed subset of tuple-distributive elements in  $2^{B_C}$ . We have to prove that  $T_P^n(\text{lub}(X)) = \text{lub}(\{T_P^n(I) \mid I \in X\})$ . Now

$$A \in T_P^n(\text{lub}(X))$$

iff there exists

$$A_1 \in B_P, \quad A_1 \leftarrow B_{(1,1)}, B_{(1,2)}, \dots, B_{(1,n_1)} \ll C_1, \quad C_1 \in P,$$

$$A_2 \in B_P, \quad A_2 \leftarrow B_{(2,1)}, B_{(2,2)}, \dots, B_{(2,n_2)} \ll C_2, \quad C_2 \in P,$$

$\vdots$

$$A_k \in B_P, \quad A_k \leftarrow B_{(k,1)}, B_{(k,2)}, \dots, B_{(k,n_k)} \ll C_k, \quad C_k \in P,$$

s.t.

$$B_{(1,1)}, B_{(1,2)}, \dots, B_{(k,n_k)} \in \text{lub}(X) \quad \text{and} \quad A \in \alpha(A_1, \dots, A_k)$$

iff there exists

$$A_1 \in B_P, \quad A_1 \leftarrow B_{(1,1)}, B_{(1,2)}, \dots, B_{(1,n_1)} \ll C_1, \quad C_1 \in P,$$

$$A_2 \in B_P, \quad A_2 \leftarrow B_{(2,1)}, B_{(2,2)}, \dots, B_{(2,n_2)} \ll C_2, \quad C_2 \in P,$$

$\vdots$

$$A_k \in B_P, \quad A_k \leftarrow B_{(k,1)}, B_{(k,2)}, \dots, B_{(k,n_k)} \ll C_k, \quad C_k \in P,$$

s.t.

$$B_{(1,1)}, B_{(1,2)}, \dots, B_{(k,n_k)} \in I \quad \text{for some} \quad I \in X \quad \text{and} \quad A \in \alpha(A_1, \dots, A_k)$$

iff

$$A \in T_P^n(I) \quad \text{for some} \quad I \in X$$

iff

$$A \in \text{lub}(\{T_P^n(I) \mid I \in X\}). \quad \square$$

*Result.* Let  $P$  be a logic program. Then the least fixpoint of  $T_P^n$  is  $T_P^n \uparrow \omega$ , and is denoted by  $\llbracket P \rrbracket^n$ .

*PROOF.* By Tarski and Kleene [19, 9] we know that if  $T: C \rightarrow C$  is continuous, where  $C$  is a complete lattice, then  $\text{lfp}(T) = T \uparrow \omega$ .

<b>Input:</b>	A logic program $P$ A ground goal $G$
<b>Output:</b>	true if $G \in \llbracket P \rrbracket^a$
<b>Interpreter:</b>	The initial goal consists of $\text{paths}(G)$ . While the goal is not empty do Choose a path $\xi$ from the goal and choose $A \leftarrow B_1, \dots, B_m \leftarrow C, C \in P$ s.t. $\xi \in \text{paths}(A)$ Remove $\xi$ from the goal and add $\text{paths}(\{B_1, \dots, B_m\})$ to the goal. If the goal is empty output true.

FIGURE 2. The abstract interpreter.

### 3.2. The Abstract Interpreter

We present an interpreter (Figure 2) that recognizes goals that are in  $\llbracket P \rrbracket^a$ . The goal contains a set of ground paths. The interpreter nondeterministically finds a successful execution if one exists.

*Lemma 3.*  $G \in \llbracket P \rrbracket^a$  iff the above interpreter has an execution that outputs true.

**PROOF.**  $\Rightarrow$ : If  $G \in \llbracket P \rrbracket^a$ , then there exists  $n$  s.t.  $G \in T_P^n \uparrow n$ . The proof is by induction on  $n$ . If  $n = 1$ , then  $\text{paths}(G) \subseteq \text{paths}(\{X \mid X \text{ is a fact in } P\})$ . Since the body of facts is empty, then every iteration of the interpreter reduces a path from the goal, and that implies that it will be empty after  $|\text{paths}(G)|$  iterations. Assume true for  $n - 1$ , and prove for  $n$ . There exist

$$\begin{aligned} A_1 \in B_P, \quad A_1 \leftarrow B_{(1,1)}, B_{(1,2)}, \dots, B_{(1,n_1)} \leftarrow C_1, \quad C_1 \in P, \\ A_2 \in B_P, \quad A_2 \leftarrow B_{(2,1)}, B_{(2,2)}, \dots, B_{(2,n_2)} \leftarrow C_2, \quad C_2 \in P, \\ \vdots \\ A_k \in B_P, \quad A_k \leftarrow B_{(k,1)}, B_{(k,2)}, \dots, B_{(k,n_k)} \leftarrow C_k, \quad C_k \in P, \end{aligned}$$

s.t.

$$B_{(1,1)}, B_{(1,2)}, \dots, B_{(k,n_k)} \in T_P^{n-1} \uparrow (n-1) \quad \text{and} \quad G \in \alpha(\{A_1, \dots, A_k\}).$$

After  $|\text{paths}(G)|$  iterations the goal can contain exactly  $\text{paths}(\{B_{(1,1)}, \dots, B_{(k,n_k)}\})$ . By the induction hypothesis each of the  $B_{(i,j)}$  is "provable" by the interpreter. Hence, after a finite number of iterations the interpreter answers true.

$\Leftarrow$ : In this part we prove that if the goal is initialized to a set  $S$  and the interpreter outputs true, then  $S \subseteq \text{paths}(\llbracket P \rrbracket^a)$ . This implies the lemma. The proof is by induction on the number,  $n$ , of iterations that the interpreter does. If  $n = 0$  then  $\emptyset = S \subseteq \text{paths}(\llbracket P \rrbracket^a)$ . Assume true for  $n - 1$  and prove for  $n$  iterations. Also assume that  $\xi$  is the first path chosen in the first iteration and that  $A \leftarrow B_1, \dots, B_m \leftarrow C, C \in P$ , is the chosen clause s.t.  $\xi \in \text{paths}(A)$ . By the induction hypothesis

$$S_1 = ((S \setminus \{\xi\}) \cup \text{paths}(\{B_1, \dots, B_m\})) \subseteq \text{paths}(\llbracket P \rrbracket^a)$$

If we apply  $T_P^\alpha$ , which is monotonic, to both sides of the inclusion, we get that

$$\xi \in \text{paths}(A) \subseteq T_P^\alpha(S_1) \subseteq \text{paths}([P]^\alpha).$$

Finally, we get that

$$S \subseteq (S_1 \cup \{\xi\}) \subseteq \text{paths}([P]^\alpha). \quad \square$$

*Example.* Let  $P$  be the following program:

```
q(c) ← p(a, d).
p(a, b).
p(c, d).
```

Then

$$[P] = \{p(a, b), p(c, d)\},$$

$$\alpha([P]) = \{p(a, b), p(a, d), p(c, b), p(c, d)\} = \alpha(T_P \uparrow 1) = T_P^\alpha \uparrow 1,$$

$$[P]^\alpha = \{p(a, b), p(a, d), p(c, b), p(c, d), q(c)\} = T_P^\alpha \uparrow 2.$$

One should note that  $[P]^\alpha$  is not always  $\alpha([P])$ , nor always a fixpoint of  $T_P$ , but it is always true that  $[P] \subseteq \alpha([P]) \subseteq [P]^\alpha$ .

Observe that  $T_P^\alpha$  is the optimal approximation function of  $T_P$  with respect to the domain, i.e.  $T_P^\alpha = \alpha \circ T_P \circ \gamma$ . More details can be found in [4].

#### 4. WELL-TYPING

*Definition.* Let  $P$  be a logic program and let  $S$  be a type. Then:

$P$  is *weakly well typed* by  $S$  iff it does not contain useless clauses relative to  $S$ .

$P$  is *well typed* by  $S$  iff it is weakly well typed by  $S$  and  $T_P^\alpha(S) = S$ , i.e.,  $S$  is a fixpoint of  $T_P^\alpha$ .

We define a program to be well typed with respect to a type if the type is *any* fixpoint of the abstract operator. It is shown below that such types form useful approximations to the meaning of a program and declaring them is not difficult.

#### 5. TYPE DECLARATIONS

Usually a programmer has in mind the type of arguments that a procedure is to be used with, but in fact the program actually written accepts larger types. For example consider the program

```
plus(0, Y, Y).
plus(s(X), Y, s(Z)) ←
  plus(X, Y, Z).
```

The type of this program is usually define to be the set

$$\{\text{plus}(X, Y, Z) \mid X, Y, Z \in \{0, s(0), s(s(0)), \dots\}\}.$$

But the program's meaning is a superset of this type if  $L$  contains constants or

function symbols other than 0,  $s/1$ . This happens because the second and third arguments of the first clause are not restricted to have the appropriate set of values. This leads to the idea of ensuring that the program can derive only the desired ground atoms.

*Definition.* A variable in a clause is a *head-only variable* if it occurs in its head but not in its body.

If a program  $P$  has no head-only variables, then only ground goals that are formed from  $\text{sig}(P)$  are derivable from the program by resolution. It should be also noted that all the facts in such a program are ground.

*Lemma 4.* If  $P$  is a logic program without head-only variables, then the set of atoms derivable from  $P$  using resolution is ground.

PROOF. By induction on  $n$ , the length of the derivation.  $\square$

We will use the type declarations to eliminate head-only variables by a transformation that adds a type condition to the body of a clause for every occurrence of such variable. A similar idea was suggested by Naish [17], where types are checked also at run time.

In the first subsection we describe a language for declaring regular types. In the second subsection we give examples of well-typed programs.

### 5.1. Regular Type Declarations

Since regular types seem sufficiently expressive and easy to manipulate, we restrict ourselves to regular type declarations. We use the elegant BNF derivation rules to represent RUL programs. A type declaration induces a type on each argument position (node) of an atom that appears in the program, including the head-only variables. For each such variable we add to the body of the clause a predicate that verifies that the variable's value is in the induced type.

*Example.* A program with a type declaration:

```
Natural ::= 0 ; s(Natural).
Binary_tree ::= void ; tree(Natural, Binary_tree, Binary_tree).
Procedure isotree(Binary_tree, Binary_tree).
isotree(void, void).
isotree(tree(X, Left1, Right1), tree(X, Left2, Right2)) ←
  isotree(Left1, Left2),
  isotree(Right1, Right2).
isotree(tree(X, Left1, Right1), tree(X, Left2, Right2)) ←
  isotree(Left1, Right2),
  isotree(Right1, Left2).
```

The transformed program with the type declaration and explicit type conditions is

```
Natural ::= 0 ; s(Natural).
Binary_tree ::= void ; tree(Natural, Binary_tree, Binary_tree).
Procedure isotree(Binary_tree, Binary_tree).
```

```

isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  natural(X),
  isotree(Left1,Left2),
  isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left1,Right2)) ←
  natural(X),
  isotree(Left1,Right2),
  isotree(Right1,Left2).
Procedure natural(Natural).
natural(0).
natural(s(X)) ←
  natural(X).
Procedure binary_tree(Natural,Binary_tree,Binary_tree).
binary_tree(void).
binary_tree(X,Left,Right) ←
  natural(X),
  binary_tree(Left),
  binary_tree(Right).

```

The derivation rule is the same as in BNF. The starting symbol is *Procedure*, where *Procedure pred* is a shorthand for *Procedure ::= pred*. The “;” symbol is used instead of the “|” symbol in BNF.

The set  $S$  that correspond to the type declaration of the program is the set that is derived from the nonterminal *Procedure*.

The example above shows a typed program, which is well typed. The isomorphism between the RUL type definition and the BNF definition is trivial. The syntax for type definitions has the following form:

$$\begin{aligned}
 A_1 &::= \alpha_{1,1}; \alpha_{1,2}; \dots; \alpha_{1,m_1} \\
 A_2 &::= \alpha_{2,1}; \alpha_{2,2}; \dots; \alpha_{2,m_2} \\
 &\vdots \\
 A_n &::= \alpha_{n,1}; \alpha_{n,2}; \dots; \alpha_{n,m_n} \\
 \textit{Procedure } &\beta_1 \\
 &\text{Code of procedure } \beta_1 \\
 \textit{Procedure } &\beta_2 \\
 &\text{Code of procedure } \beta_2 \\
 &\vdots \\
 \textit{Procedure } &\beta_m \\
 &\text{Code of procedure } \beta_m
 \end{aligned}$$

where:

The  $A_i$ 's are variables, and the  $\alpha_{i,j}$ 's are nonvariable terms.

For every  $1 \leq i \leq n$ ,  $1 \leq j < k \leq m_i$ ,  $\alpha_{i,j}$  and  $\alpha_{i,k}$  are top-level not unifiable.

The  $\beta_i$ 's are atoms with different predicates.

An alternative definition of the semantics of a typed program has been considered in an earlier version of this paper [23]. The idea was to restrict all the variables to their induced type. This means that a larger variety of nonintuitive type declarations could well-type the program. For instance the program

```
Nat01 ::= 0 ; s(0).
procedure plus(Nat01, Nat01, Nat01).
plus(0, X, X).
plus(s(X), Y, s(Z)) ←
  plus(X, Y, Z).
```

is well typed by the earlier definition, but not by the current one.

## 5.2. Examples

Two more examples that are long enough and use a nontrivial variety of data structures are presented, demonstrating the ease of well-typing.

*Example (The towers-of-Hanoi problem).* Given three pegs labeled  $a$ ,  $b$ , and  $c$ . The problem is to move a tower of  $n$  disks from peg  $a$  to peg  $c$ , with the help of peg  $b$ . Only one disk can be moved at a time, and a larger disk can never be placed on top of a smaller disk.

```
Natural ::= 0 ; s(Natural).
Moves ::= [] ; [Move | Moves].
Diff_Moves ::= Moves \ Moves.
Move ::= (Peg, Peg).
Peg ::= a ; b ; c.

procedure hanoi(Natural).
hanoi(N) ← hanoi(N, X).

procedure hanoi(Natural, Moves).
hanoi(N, X) ← hanoi(N, a, c, X \ []).

procedure hanoi(Natural, Peg, Peg, Diff_Moves).
hanoi(0, From, To, X \ X).
hanoi(s(N), From, To, Before \ Tail) ←
  free(From, To, Free),
  hanoi(N, From, Free, Before \ [(From, To) | After]),
  hanoi(N, Free, To, After \ Tail).

procedure free(Peg, Peg, Peg).
free(a, b, c). free(a, c, b). free(b, a, c).
free(b, c, a). free(c, a, b). free(c, b, a).
```

*Example (Priority queue).* In the following example a priority queue is represented as a list of pairs  $(X, P)$ , where  $X$  is the element and  $P$  is its associated priority. On  $enqueue(X, P)$  the queue process inserts  $X$  into the list according to its priority; on  $dequeue(X)$  it moves  $X$  from the head of the list. Note that  $any$  is just a regular constant (see also discussion of the supertype  $Any$  in Section 7).

```

Natural ::= 0 ; s(Natural).
Enqueue ::= [] ; [enqueue(any, Natural) | Enqueue].
Dequeue ::= [] ; [dequeue(any) | Dequeue].
Queue ::= [] ; [(any, Natural) | Queue].
procedure Natural ≤ Natural.
0 ≤ Y.
s(X) ≤ s(Y) ← X ≤ Y.
procedure Natural < s(Natural).
X < s(Y) ← X ≤ Y.
procedure queue(Enqueue, Dequeue).
queue(Es, Ds) ← queue(Es, Ds, []).
procedure queue(Enqueue, Dequeue, Queue).
queue(Es, [dequeue(X) | Ds], [(X, P) | Q]) ←
  queue(Es, Ds, Q).
queue([enqueue(X, P) | Es], Ds, Q) ←
  insert((X, P), Q, Q1),
  queue(Es, Ds, Q1).
queue([], [], Q).
procedure insert((any, Natural), Queue, [(any, Natural) | Queue]).
insert((X, P), [(X1, P1) | Q], [(X, P), (X1, P1) | Q]) ←
  P ≤ P1.
insert((X, P), [(X1, P1) | Q], [(X1, P1) | Q1]) ←
  P1 < P,
  insert((X, P), Q, Q1).
insert((X, P), [], [(X, P)]).

```

## 6. TYPE CHECKING

Type checking determines whether a program is well typed by a type declaration. Type checking in our framework requires checking inclusion in a type, which is undecidable for r.e. types. Therefore we investigate regular types, for which type checking is decidable. It seems that regular types are strong enough for type declarations. We suggest an algorithm for type checking and analyze its complexity.

### 6.1. A Type-Checking Algorithm

We describe an algorithm (Figure 3) that type-checks a typed program whose type is declared by a regular type declaration.

Let  $P$  be a program and  $S$  be a regular type. We check whether  $P$  is well typed by  $S$  as follows: For each clause in  $P$  we find the set of atoms that can be inferred relative to  $S$ . Then we find the union of all these sets and check if it equals  $S$ .

The type checker should also report on clauses that are useless, i.e., if there is no ground instance of the clause that is constructed from the type.

The key idea in focusing on regular types, which are represented by DFA, is that it is possible to infer the maximal set of values that a node of an atom can obtain with respect to the type declaration  $S$  and hence the maximal set of values that a variable can assume (we call this *the type of the variable*).

```

For each clause  $C$  in  $P$  do:
  For every ground path  $x$  in the body of  $C$  do:
    If  $x \notin paths(S)$  then
       $T_{(C)}^a(S) := \emptyset$ 
      goto 1.
    endif
  od
  For each variable  $Y$  that appears in  $C$  do:
    For each occurrence of  $Y$  in the body of  $C$  do:
      Infer the maximal set of values that this occurrence can obtain.
    od
    Intersect all the above sets getting the variable's type.
  od
  Construct  $T_{(C)}^a(S)$  using the type of the variables.
  1: If  $T_{(C)}^a(S) = \emptyset$  then print a warning that clause  $C$  is useless.
od
Find the tuple-distributive closure of the union of the  $T_{(C)}^a(S)$ 's computed.
If the result is equal to  $S$  and no clause is useless then succeed else fail.

```

FIGURE 3. A type-checking algorithm.

We want to show that we can compute  $paths(T_{(C)}^a(S))$ . For each node in the body of the clause  $C$  where a variable appears, we can find the maximal set of values that it can obtain. If  $B$  is an atom in  $C$ , and  $Y$  is some node of  $B$ , then we can find the path from the root to  $Y$  in the associated tree of  $B$ , and run this path on the DFA that represents  $S$  getting to some state  $q$ . The set of all strings that lead from  $q$  to a final state represents the maximal set of values that  $Y$  can obtain with respect to  $S$ . All nodes in the body of  $C$  that have the same logical variable must be intersected to obtain its type. Unnecessary paths, i.e., paths that cannot correspond to a term, are assumed to be deleted. If there is a variable in  $C$  that has an empty type or there is a ground path of some atom in the body of  $C$  that is not in  $paths(S)$ , then  $T_{(C)}^a(S) = \emptyset$ , else we can build the automaton for  $paths(T_{(C)}^a(S))$  from the head of the clause and the types that were found for the variables in the body of the clause.

*Claim.* Let  $S$  be a regular type. Then  $P$  is well typed by  $S$  iff the algorithm in Figure 3 succeeds.

*PROOF.* We have to prove that  $T_P^a(S) = S$  iff the algorithm succeeds:

$$\begin{aligned}
T_P^a(S) &= \alpha(\{A \in B_L \mid A \leftarrow B_1, B_2, \dots, B_n \leftarrow C, C \in P, \text{ and } B_1, \dots, B_n \in S\}) \\
&= \alpha\left(\bigcup_{C \in P} \{A \in B_L \mid A \leftarrow B_1, \dots, B_n \leftarrow C \text{ and } B_1, \dots, B_n \in S\}\right) \\
&= \left\{X \mid paths(X) \subseteq \bigcup_{C \in P} paths(T_{(C)}^a(S))\right\}.
\end{aligned}$$

So we get that  $T_P^a(S) = S$  iff  $\bigcup_{C \in P} paths(T_{(C)}^a(S)) = paths(S)$ . We have shown above that finding  $T_{(C)}^a(S)$  is possible. The union of the  $T_{(C)}^a(S)$ 's is simply the union of DFA, getting possibly a nondeterministic finite automaton (NFA).  $\square$

In our implementation we transform the NFA that was constructed to a DFA, which may take exponential time, and then, by an equivalence algorithm which is known to be almost linear, we do the last step of the algorithm.

### 6.2. Complexity analysis

We now consider the complexity of the above algorithm. We show that the complexity is exponential in the maximal number of clauses that belong to the same procedure and is exponential in the maximal number of occurrences of a variable in a clause.

Assume that:

$N_p$  is the number of different predicates in  $P$ .

$N_v$  is the maximal number of all occurrences of a variable in a clause of  $P$ .

$L_c$  is the maximal length of a clause in  $P$ .

$N_p$  is the maximal number of clauses of the same predicate.

$N$  is the number of states in the DFA that represents the type.

Then:

Inferring the maximal possible set of values of a variable occurrence in a clause requires finding the right state in the DFA, which can be done in time  $O(NL_c)$ .

Intersection of the regular sets to get the type of a variable takes  $O(N^{N_v})$ .

Constructing  $T_{(C)}^g(S)$  can be done in  $O(L_c N^{N_v})$ .

Uniting two DFAs and transforming their union into an equivalent DFA takes  $O(N^2)$ . So uniting the  $T_{(C)}^g(S)$ 's to get a DFA takes  $O(N_p(N_p L_c N^{N_v} + N^{N_v N_p}))$ .

The equivalence of the DFA that represents the type and the DFA that was constructed is almost linear. So the overall complexity of the algorithm is  $O(N_p(N_p L_c N^{N_v} + N^{N_v N_p}))$ .

In practice, the number of intersections in a clause is small, and in most applications that do not involve databases the number of clauses in a procedure is small. In all programs that appear in this paper the number of occurrences of the same logical variable does not exceed four. The intersection of these occurrences does not take exponential time, because they all lead to the same state in the DFA that represents the type.

If we have to deal with a large database of ground facts, then the algorithm takes polynomial time.

It is possible to define a subset of logic programs for which the algorithm has polynomial-time complexity:

- (a)  $N_v$  is bounded by a constant.
- (b) If  $h$  and  $h'$  are heads of different clauses and have the same predicate, then for all  $i$ , if  $arg_i$  is the  $i$ th argument of  $h$  and  $arg'_i$  is the  $i$ th argument of  $h'$ , then  $arg_i$  and  $arg'_i$  are top-level not unifiable.

It seems that (a) is a reasonable demand but (b) is too restrictive, as it excludes most deterministic and many other programs.

## 7. IMPLEMENTATION

A type checker for FCP based on this framework has been implemented in FCP and is incorporated in the Logix system [18]. Each clause is stripped of control symbols. The type of the guards and the system predicates are predefined. Each program is checked procedurewise; each clause of a procedure is checked for usability, and each procedure is checked for the fixpoint requirement. If an atom does not conform to the type definition, then an error consisting of the clause number and the leftmost path that doesn't agree with the definition is displayed. If the intersection of variable's types is empty, then the name of the variable is displayed. If the inferred type is different from the defined type, then an error with one path in the difference set is displayed.

Although type declarations and checking are not required in Logix, most large applications developed under Logix use the type system.

Practice with the type system has shown that its precision is sometimes a nuisance. Sometimes a programmer does not want to specify precisely the integer values an argument can take, but just that it takes integer values, and similarly for strings. Sometimes a process participating in a stream communication protocol may wish to send only a subset of the messages the receiver can handle, but not to specify precisely that subset. To that effect we have incorporated a subset typing mechanism. One type can be defined as a subtype of another, using the notation

*SomeValue* ::< *Integer*.

*KeyWords* ::< *String*.

etc.

Similarly, an all-out escape is provided, in terms of the supertype *Any* that corresponds to the Herbrand universe. The declaration

*T* ::< *Any*.

means that *T* can take any value. In contrast, the declaration

*T* ::= *Any*.

means that *T* must be able to take all values. The former is a well-typing of any predicate. Given the type declaration *procedure p(T)*, the latter definition of *T* is a well-typing of the procedures  $\{p(X)\}$ ,  $\{p(X), p(a)\}$ , but not of  $\{p(a)\}$  or of  $\{p(f(X))\}$ .

## 8. DISCUSSION

From our experience it seems that a regular type can be constructed naturally for every program. The programmer knows the type that each of its variables can obtain, and that makes it quite easy to declare types. When we use large databases it may be cumbersome to declare all the values that an argument can obtain, since the list would be a long declaration. However, even in this case the declaration is still conceptually easy.

Our type system deals only with pure logic programs that are self-contained. This is the basic theoretical model from which we build type systems for concrete logic programming languages such as FCP or PROLOG. In such languages there are system predicates and procedures like *append/3*, which appends two lists that may contain any sort of data. Effective typing of such programs requires parametric types. We should also note that a module of a program that is not self-contained cannot be well typed, since the fixpoint requirement is violated. This problem can be solved by requiring the fixpoint procedurewise. Another construct that other languages have is basic types like integers, strings, atoms, etc., which are a necessity in type systems. This implies that:

It is necessary to extend the existing type theory to parametric types. This is a subject of a subsequent paper [24].

The theory should be extended to deal with modules and basic types.

The relation between type checking, well-typing, and type inference deserves further investigation. Further research includes an algorithm for inferring the minimal type of the program and extending the definition of well-typing to partially declared programs.

## APPENDIX. RUL PROGRAMS

### A.0. Introduction

Canonical RUL programs are presented, and a transformation algorithm from RUL programs to canonical RUL programs which preserves the relative meaning of the program is described. The relation between the meaning of canonical RUL programs and DFAs is discussed, and the implementation of regular types as DFAs is shown. Thus the identity of the meaning of RUL programs and regular types is established.

*Definition.* The meaning of a clause  $C$  in a program  $P$  is the set

$$\{A \in B_p \mid A \leftarrow B_1, \dots, B_n \leftarrow C, B_1, \dots, B_n \in [P]\}.$$

*Example.* The meaning of a ground fact is the set that contains only the ground fact.

*Definition.* A clause is *nonredundant* with respect to a program if its meaning is nonempty. It is *redundant* if its meaning is empty.

*Definition.* A *nonredundant* program is a program with no redundant clauses.

In general, it is undecidable whether a program is nonredundant, but we will see that for RUL programs this problem is decidable.

*Claim.* For every RUL program  $P$ , there exists a nonredundant RUL program  $Q$  s.t.  $[P] = [Q]$ .

PROOF. Delete the redundant clauses of  $P$  to obtain  $Q$ .  $\square$

*A.0.1. Determining Redundancy of Clauses.* This subsection shows how to find redundant clauses of RUL programs and to find a concise representation of the equivalent nonredundant RUL program.

*A.0.2. The Algorithm.* Let  $P$  be a RUL program.

1. Label all facts in  $P$  as nonredundant, and call this group  $A_0$ .

2. Let  $S_0 = \{p \mid p \text{ is a predicate name of a fact in } P\}$ .

3. Let  $S_i$  be the last set that was computed and define  $A_{i+1}$  and  $S_{i+1}$ :

$$A_{i+1} = \{C \in P \mid C = p(t) \leftarrow b_1(X_1), \dots, b_n(X_n) \text{ unlabeled and } b_1, \dots, b_n \in S_i\}$$

$$S_{i+1} = S_i \cup \{p \mid p \text{ is a predicate name that appears in } A_{i+1}\}.$$

4. Label all clauses in  $P$  that appear in  $A_{i+1}$  as nonredundant.

5. If  $S_i = S_{i+1}$  then stop  
else go to 3.

*A.0.3. Correctness of the Algorithm.*

*Claim.* All clauses in  $P$  that were marked by the above algorithm are nonredundant.

PROOF. The proof is by induction on  $i$  that all the clauses in  $A_i$  are nonredundant, and this will imply the claim.

$i = 0$ : Since the meaning of a fact is nonempty, the claim holds.

Assume the claim holds for less than  $n$ , and prove for  $n$ : Let  $p(t) \leftarrow b_1(X_1), \dots, b_m(X_m)$  be a clause in  $A_n$ . By the definition of  $A_n$ , we have  $b_1, \dots, b_m \in S_{n-1}$ . This means that for all  $j \in [1..m]$  there exists a labeled clause whose head is  $b_j$ . By the induction hypothesis those clauses are nonredundant. It follows that  $b_j(t_1), \dots, b_m(t_m) \in [P]$  for some ground terms  $t_1, \dots, t_m$ . By the definition of the meaning of a clause and by the characteristics of a clause of a RUL program,  $p(t) \leftarrow b_1(X_1/t_1), \dots, b_m(X_m/t_m)$  is ground and in the meaning of the above clause.  $\square$

For the next two claims assume that stage 5 of the algorithm is just "goto 3".

*Claim.* If  $S_n = S_{n-1}$  for some  $n$ , then for all  $i \geq n$ ,  $S_i = S_n$ .

PROOF. Trivial.  $\square$

*Claim.* All the predicate names that appear in  $T_P \uparrow i + 1$  also appear in  $S_i$ .

PROOF. Easy induction on  $i$ .  $\square$

*Claim.* All the predicate names that are in the meaning of the program are also in the last set  $S_i$  of the algorithm.

PROOF. The meaning of a program is  $T_P \uparrow \omega$ . All the predicate names that appear in the meaning of the program also appear in  $S_\omega$ . But  $S_\omega = S_i$  by the claims above.  $\square$

*Claim.* All the clauses that were not labeled by the above algorithm are redundant.

*PROOF.* By contradiction. Assume that there exists a nonredundant clause  $p(t) \leftarrow b_1(X_1), \dots, b_n(X_n)$  that was not marked by the algorithm. Then, all the predicates of the  $b_j$ 's appear in the meaning of  $P$ . By the previous claim, these predicate names appear also in  $S_{i+1}$ . By the algorithm, it is clear that the  $b_j$ 's also appear in  $S_j$ . Then by steps 3 and 4, before the  $(i+1)$ th iteration, the above clause is labeled. Contradiction.  $\square$

*RESULT.* Let  $P$  be a RUL program. Then a clause is nonredundant iff it was labeled by the above algorithm.

#### A.1. Canonical RUL Programs

*Definition.* A canonical RUL program is a nonredundant RUL program satisfying the following:

Arguments of facts are constants.

If the head argument of a clause is a compound term, i.e. a function symbol applied to arguments, then all of the arguments are variables.

The example of a RUL program in subsection 2.4 satisfies the syntactic conditions for canonical RUL programs, but it is redundant, since the predicate *natural/1* is missing.

*Notation.* Let  $M = \langle Q, \Sigma, \delta, S, \{F\} \rangle$  be a DFA. For every  $q \in Q$ , denote

$$\mathcal{L}_q(M) = \left\{ s \mid (q, s) \vdash_M^* (F, \epsilon) \right\}.$$

*REMARK.*  $\mathcal{L}_q(M)$  denotes the regular language accepted by the same automata as  $M$ , starting from the state  $q$ .

*Lemma A.1.* Canonical RUL programs are regular.

*PROOF.* Let  $P$  be a canonical RUL program. Build a DFA  $M = \langle Q, \Sigma, \delta, S, \{F\} \rangle$  according to the following:

$Q = \{p \mid p \text{ is a predicate in } P\} \cup \{S, F\}$ .

$F$  = final state, where  $F$  is different from  $S$ .

$\Sigma = \Sigma_P$ .

For every predicate  $q$  in  $P$ ,  $\delta(S, q(1, 1)) = q$ .

For each clause of a predicate  $p$  define  $\delta$  to be the following:

- (1) If the heads argument is a constant  $c$  then  $\delta(p, c) = F$ .
- (2) If the head argument is a function symbol  $f$  with arity  $n$ , then for every  $i \in [1..n]$ ,  $\delta(p, f(n, i)) = q$ , where the body goal  $q$  and the  $i$ th argument of the function share the same variable.

REMARK.  $M$  is a DFA, since every head argument of a procedure clause has a unique function-symbol-arity combination.

Claim A.1.  $t \in [P]/p$  iff  $paths(t) \subseteq \mathcal{L}_p(M)$ .

The proof of this claim follows the end of this lemma.  
Back to the proof of the lemma.

$p(t) \in [P]$

iff

$paths(t) \subseteq \mathcal{L}_p(M)$

iff

$paths(p(t)) \subseteq \mathcal{L}(M)$ .  $\square$

PROOF OF CLAIM A.1. By induction on  $|t|$ : Let

$|t| = 1$ .

That means  $t = c$  for some constant  $c$ . Then

$t \in [P]/p$

iff (by definition)

$p(t) \in [P]$

iff (by the semantics of logic programs and structure of  $P$ )

there exists a fact  $p(c)$  in  $P$

iff (by the construction of  $M$ )

$\delta(p, c) = F$

iff (by definition)

$c \in \mathcal{L}_p(M)$

iff ( $paths(c) = \{c\}$ )

$paths(t) \subseteq \mathcal{L}_p(M)$ .

Assume the claim is true for all  $t$ ,  $0 < |t| < N$ , and prove for  $|t| = N$ . W.l.o.g.  $t = f(t_1, t_2, \dots, t_n)$ . Then

$p(t) \in [P]$

iff (by the semantics of logic programs)

there exists a clause in  $P$  of the form

$p(f(X_1, X_2, \dots, X_n)) \leftarrow b_1(X_1), \dots, b_n(X_n)$ ,

and for every  $j \in [1..n]$ ,  $b_j(t_j) \in [P]$

iff (by the construction of  $M$  and by the induction hypothesis)

for every  $j \in [1..n]$ ,  $\delta(p, f(n, j)) = b_j$  and  $paths(t_j) \subseteq \mathcal{L}_{b_j}(M)$

iff (by the characteristics of DFAs and  $M$ )

$\{f(n, j) \cdot s_j \mid s_j \in paths(t_j), j \in [1..n]\} \subseteq \mathcal{L}_p(M)$

iff (by the characteristics of *paths*)

$$\text{paths}(t) \subseteq \mathcal{L}_p(M). \quad \square$$

*Lemma A.2.* For every regular set of terms  $T$ , there exists a canonical RUL program  $P$  with a predicate  $p$  s.t.  $\llbracket P \rrbracket/p = T$ .

*PROOF.*  $T$  regular means that there exists a DFA (w.l.o.g. the starting state is  $p$ )  $M = (Q, \Sigma, \delta, p, \{F\})$  s.t. for every term  $t$ ,  $t \in T$  iff  $\text{paths}(t) \subseteq \mathcal{L}(M)$ . Build a program  $P$  according to the following: For every state  $q$  and every function symbol  $f$ , if  $\delta(q, f(n, i)) = b_i$  is defined for all  $i \in [1..n]$ , then add the clause  $p(f(X_1, X_2, \dots, X_n)) \leftarrow b_1(X_1), \dots, b_n(X_n)$  to  $P$ . For every state  $q$  and a constant  $c$ , if  $\delta(q, c)$  is a final state add the clause  $q(c)$  to  $P$ .

Observations:

- (1)  $P$  is a canonical RUL program (it might have redundant clauses, but they can be deleted).
- (2) Almost by the same proof as that of Claim A.1,  $t \in \llbracket P \rrbracket/q$  iff  $\text{paths}(t) \subseteq \mathcal{L}_q(M)$ .
- (3) By observation (2) and by regularity,  $t \in T$  iff  $t \in \llbracket P \rrbracket/p$ .  $\square$

*Lemma A.3.* For every RUL program  $P$ , there exists a canonical RUL program  $Q$  s.t.  $\llbracket P \rrbracket = \llbracket Q \rrbracket \setminus \{q(t) \in \llbracket Q \rrbracket \mid q \text{ is a predicate not occurring in } P\}$ .

*PROOF.* The idea is to unravel the head arguments. W.l.o.g.  $P$  is nonredundant. Define a program transformation,  $Tr(R)$ , on a RUL program  $R$  as follows: If there exists a clause in  $R$  of the form

$$p(f(\text{Arg}_1, \dots, \text{Arg}_n)) \leftarrow b_1(X_1), \dots, b_k(X_k) \in P,$$

and  $\exists j \in [1..n]$  s.t.  $\text{Arg}_j$  is not a variable, then do the following:

Find a new predicate symbol  $q$  and a new variable symbol  $X$ . W.l.o.g.  $X_1, \dots, X_n$  are all the variables appearing in  $\text{Arg}_j$ .

Transform this clause into two clauses:

$$\begin{aligned} & p(f(\text{Arg}_1, \dots, \text{Arg}_{j-1}, X, \text{Arg}_{j+1}, \dots, \text{Arg}_n)) \\ & \quad \leftarrow q(X), b_{m+1}(X_{m+1}), \dots, b_k(X_k). \\ & q(\text{Arg}_j) \leftarrow b_1(X_1), \dots, b_m(X_m). \end{aligned}$$

Perform the above transformation on  $P$ , getting a sequence  $P = P_1, P_2, \dots, P_k = Q$ , where  $P_{i+1} = Tr(P_i)$  and  $Q$  does not contain any more clauses to be transformed.

*Claims (Easy to prove).*

For every predicate  $p$  in  $R$ , the transformation preserves  $\llbracket R \rrbracket/p$ , and nonredundancy.

The number  $k$  is bounded.

$Q$  is a canonical RUL program.

For all predicates  $p$  in  $P$ ,  $\llbracket P \rrbracket/p = \llbracket Q \rrbracket/p$ .  $\square$

### A.2. Properties of Regular Sets of Terms

*Claim.* The intersection of regular sets of terms is regular.

*PROOF.* Trivial, by the fact that the intersection of regular languages is regular.  $\square$

*Lemma A.4.* For every regular set of terms  $T$ ,  $paths(T)$  is regular.

*PROOF.* If  $T$  is regular, then there exists a predicate  $p$  in a canonical RUL program,  $P$ , s.t.  $T = \llbracket P \rrbracket / p$ . Build a DFA  $M$  as in Lemma A.1 corresponding to  $P$ . The proof is by induction that for every word  $s \in \mathcal{L}_q(M)$ , where  $q$  is a predicate in  $P$ , there exists a term  $t \in \llbracket P \rrbracket / q$  s.t.  $s$  is a path in  $t$ . This will imply the lemma.

If  $|s| = 1$ , then  $s$  is a constant. Trivial.

Assume true for  $0 < |s| < N$ , and prove for  $|s| = N$ .

W.l.o.g.  $s = f(n, i) \cdot s_j$ . By the construction of  $M$ , there exists a clause of the form  $q(f(X_1, X_2, \dots, X_n)) \leftarrow b_1(X_1), \dots, b_n(X_n)$ . Since every clause in  $P$  is nonredundant, there exist terms  $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n$  s.t. for all  $j$ ,  $t_j \in \llbracket P \rrbracket / b_j$ . By the induction hypothesis, there exists a term  $t_i \in \llbracket P \rrbracket / b_i$  s.t.  $s_j$  is a path in  $t_i$ . Gathering these facts, we get that  $s$  is a path in  $t = f(t_1, \dots, t_n)$ , where  $t \in \llbracket P \rrbracket / q$ .  $\square$

*Definition.* Let  $T$  be a regular set of terms. Define  $\mathcal{L}(T)$  to be the minimal language s.t.  $t \in T$  iff  $paths(t) \subseteq \mathcal{L}(T)$ .

*Claim.*  $\mathcal{L}(T)$  is well defined and regular.

*PROOF.* By Lemma A.4.,  $paths(T)$  is the minimal regular language that is wanted.  $\square$

*Lemma A.5.* Let  $T_1$  and  $T_2$  be regular sets of terms. Then

$$T_1 \subseteq T_2 \text{ iff } \mathcal{L}(T_1) \subseteq \mathcal{L}(T_2).$$

*PROOF.* By Lemma A.4 and the above claim.  $\square$

*Lemma A.6.* If  $T$  is a regular type and  $H :- B$  is a clause in a program  $P$ , then  $\alpha(\{H\theta \in B_p \mid B\theta \subseteq T\})$  is regular.

*PROOF.* Let  $X_1, \dots, X_k$  be all the variables in the clause. W.l.o.g.  $dom(\theta) = \{X_1, \dots, X_k\}$ . Assume that  $\theta = [X_1 \mapsto t_1, \dots, X_k \mapsto t_k]$  and that there exists an automaton  $M$  s.t.  $L(M) = paths(T)$ .

Let  $s$  be a string, and  $M = \langle Q, \Sigma, \delta, S, \{F\} \rangle$  be a DFA. Denote by  $state(s, M)$  the state  $\delta(S, s)$  if it is defined. If it is not defined, then call it  $\perp$ . Recall that  $L_q(M)$  is defined to be the language accepted by the same automaton as  $M$  except that the starting state is  $q$ . Modify the definition s.t.  $L_\perp(M) = \emptyset$ . Let  $B$  be a conjunction of one or more atoms. Define  $pos(X_j, B)$  to be the set of all paths from the root of the associated trees of  $B$  to  $X_j$ , i.e.  $\{s \mid sX_j \in paths(B)\}$ .

Define  $g\_paths(B) = \{s \mid s \in paths(B) \cap \Sigma_{\{0\}}^*\}$  to be the set of all *ground paths* in  $B$ . Then

$$\begin{aligned}
 B\theta \subseteq T &\Leftrightarrow paths(B\theta) \subseteq L(M) \\
 &\Leftrightarrow g\_paths(B) \subseteq L(M) \\
 &\wedge \bigwedge_{X_j \in vars(B)} \{ss_1 \mid s \in pos(X_j, B) \wedge s_1 \in paths(t_j)\} \subseteq L(M) \\
 &\Leftrightarrow g\_paths(B) \subseteq L(M) \\
 &\wedge \bigwedge_{X_j \in vars(B)} \forall s \in pos(X_j, B). paths(t_j) \subseteq \{s_1 \mid ss_1 \subseteq L(M)\} \\
 &\Leftrightarrow g\_paths(B) \subseteq L(M) \\
 &\wedge \bigwedge_{X_j \in vars(B)} \forall s \in pos(X_j, B). paths(t_j) \subseteq L_{state(s, M)}(M) \\
 &\Leftrightarrow g\_paths(B) \subseteq L(M) \\
 &\wedge \bigwedge_{X_j \in vars(B)} paths(t_j) \subseteq \bigcap_{s \in pos(X_j, B)} L_{state(s, M)}(M).
 \end{aligned}$$

It is easy to verify by construction of an automaton  $M1$  that  $B_p$  is regular. In the same manner we prove that

$$\begin{aligned}
 H\theta \in B_p &\Leftrightarrow g\_paths(H) \subseteq paths(B_p) \\
 &\wedge \bigwedge_{X_j \in vars(H)} paths(t_j) \subseteq \bigcap_{s \in pos(X_j, H)} L_{state(s, M1)}(M1).
 \end{aligned}$$

Let

$$\mathcal{L}(X_j) = \begin{cases} \bigcap_{s \in pos(X_j, H)} L_{state(s, M1)}(M1), & X_j \in vars(H) \setminus vars(B), \\ \left( \bigcap_{s \in pos(X_j, H)} L_{state(s, M1)}(M1) \right) \\ \cap \left( \bigcap_{s \in pos(X_j, B)} L_{state(s, M)}(M) \right), & X_j \in vars(H) \cap vars(B), \\ \bigcap_{s \in pos(X_j, B)} L_{state(s, M)}(M), & X_j \in vars(B) \setminus vars(H). \end{cases}$$

Finally we get that

$$\begin{aligned}
& H\theta \in B_p \wedge B\theta \subseteq T \\
& \Leftrightarrow g\_paths(A, B) \subseteq L(M) \\
& \wedge \bigwedge_{X_j \in \text{vars}(H) \setminus \text{vars}(B)} paths(t_j) \subseteq \bigcap_{s \in \text{pos}(X_j, H)} L_{state(s, M_1)}(M_1) \\
& \wedge \bigwedge_{X_j \in (\text{vars}(H) \cap \text{vars}(B))} paths(t_j) \subseteq \left( \bigcap_{s \in \text{pos}(X_j, H)} L_{state(s, M_1)}(M_1) \right) \\
& \quad \cap \left( \bigcap_{s \in \text{pos}(X_j, B)} L_{state(s, M)}(M) \right) \\
& \wedge \bigwedge_{X_j \in \text{vars}(B) \setminus \text{vars}(H)} paths(t_j) \subseteq \bigcap_{s \in \text{pos}(X_j, B)} L_{state(s, M)}(M) \\
& \Leftrightarrow g\_paths(A, B) \subseteq L(M) \wedge \bigwedge_{j=1}^n paths(t_j) \subseteq \mathcal{L}(X_j).
\end{aligned}$$

If one of the conjuncts above cannot be satisfied, then  $\alpha(\{H\theta \in B_p \mid B\theta \subseteq T\})$  is empty. If the formula can be satisfied, then

$$\begin{aligned}
& paths(\{H\theta \in B_p \mid B\theta \subseteq T\}) \\
& = \bigcup \{ paths(H\theta) \mid H\theta \in B_p \wedge B\theta \subseteq T \} \\
& = g\_paths(H) \cup \bigcup_{\substack{X_j \in \text{vars}(H) \\ s \in \text{pos}(X_j, H)}} \{s\} \cdot \bigcup \{ paths(t_j) \mid H\theta \in B_p \wedge B\theta \subseteq T \} \\
& = g\_paths(H) \cup \bigcup_{\substack{X_j \in \text{vars}(H) \\ s \in \text{pos}(X_j, H)}} \{s\} \cdot \bigcup \{ paths(t_j) \mid \forall 1 \leq i \leq n \ paths(t_i) \subseteq \mathcal{L}(X_i) \} \\
& = g\_paths(H) \cup \bigcup_{\substack{X_j \in \text{vars}(H) \\ s \in \text{pos}(X_j, H)}} \{s\} \cdot \bigcup \{ paths(t_j) \mid paths(X_j) \subseteq \mathcal{L}(X_j) \} \\
& = g\_paths(H) \cup \bigcup_{\substack{X_j \in \text{vars}(H) \\ s \in \text{pos}(X_j, H)}} \{s\} \cdot paths(\{t_j \mid paths(t_j) \subseteq \mathcal{L}(X_j)\})
\end{aligned}$$

It can be easily checked that  $paths(\{t_j \mid paths(X_j) \subseteq \mathcal{L}(X_j)\})$  is regular by Lemma A.4. It follows that  $paths(\{H\theta \in B_p \mid B\theta \subseteq T\})$  is regular, since it is constructed from finite concatenations and unions of regular languages.  $\square$

*Lemma A.7.* If  $P$  is a program, then for every natural number  $n$ ,  $T_p^n \uparrow n$  is regular.

**PROOF.** It can be shown by induction on  $n$  that  $T_p^n \uparrow n$  is regular, using the facts that:

The tuple-distributive closure of a finite union of regular sets of terms is a regular set.

Lemma A.6 guarantees that at each iterations of  $T_p^n$  we get a regular set.  $\square$

We thank John Gallagher, who contributed greatly to the conceptual organization of this paper and to the clarity of the presentation. We also thank Josan Jaffar and Nevin Heintze for their comments on previous drafts.

#### REFERENCES

1. Aho, A. V., Hopcroft, J. D., and Ullman, J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. Bruynooghe, M. and Janssens, G., An Instance of Abstract Interpretation Integrating Type and Mode Inferencing, in: *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, Wash., Aug. 1988, pp. 669-683.
3. Bruynooghe, M., Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs, in: *Proceedings of the First International Logic Programming Conference*, 1982, pp. 129-138.
4. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *POPL*, 1977, pp. 238-252.
5. Dietrich, R. and Hagl, F., A Polymorphic Type System with Subtypes for Prolog, in: *Proceedings of the 2nd European Symposium on Programming*, Nancy, France, Mar. 1988, Lecture Notes in Comput. Sci., Springer-Verlag, pp. 79-93.
6. Fruehwirth, T. W., Type Inference by Program Transformation and Partial Evaluation, in: *Meta-programming in Logic Programming*, MIT Press, 1989.
7. Kanamori, T. and Horiuchi, K., Type Inference in Prolog and Its Application, in: *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985, pp. 704-707.
8. Kanamori, T. and Kawamura, T., Analyzing Success Patterns of Logic Programs by Abstract Hybrid Interpretation, Technical Report TR-279, ICOT, 1987.
9. Kleene, S. C., *Introduction to Metamathematics*, Van Nostrand, 1952.
10. Kluzniak, F., Type Synthesis for Ground Prolog, in: *Proceedings of the 4th International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 788-816.
11. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1987.
12. Milner, R., A Theory of Type Polymorphism in Programming, *J. Comput. System Sci.* 17(3):348-375 (1978).
13. Mishra, P., Towards a Theory of Types in Prolog, in: *International Symposium on Logic Programming*, IEEE, 1984, pp. 289-298.
14. Magidor, M. and Moran, G., Finite Automata over Finite Trees, Technical Report 30, Hebrew Univ., Israel, 1969.
15. Mycroft, A. and O'Keefe, R. A., A Polymorphic Type System for Prolog, in: *Logic Programming Workshop*, 1983, pp. 107-121.
16. Naish, L., Specification = Program + Types, in: *Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1987, pp. 326-339.
17. Silverman, W., Hirsch, M., Hourii, A., and Shapiro, E., The Logix System User Manual Version 2.0, Technical Report CS-21, Weizmann Inst. of Science, Nov. 1988.
18. Tarski, A., A Lattice-Theoretical Fix-Point Theorem and Its Applications, *Pacific J. Math.* 5:285-309 (1955).
19. Thatcher, J. W., Tree Automata: An Informal Survey, in: Alfred V. Aho (ed.), *Currents in the Theory of Computing*, Prentice-Hall, 1973, Chapter 4, pp. 143-172.
20. van Emden, M. H. and Kowalski, R., The Semantics of Predicate Logic as a Programming Language, *J. Assoc. Comput. Mach.* 23(4):733-742 (Oct. 1976).

22. Xu, J. and Warren, D. S., A Type Inference System for Prolog, in: *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, Wash., Aug. 1988, pp. 604-619.
23. Yardeni, E., A Type System for Logic Programs, in: Ehud Shapiro (ed.), *Concurrent Prolog* MIT Press, 1987, Chapter 28; TR CS87-05, Weizmann Inst. of Science.
24. Yardeni, E., Fruchwirth, T. and Shapiro, E., Polymorphically Typed Logic Programs, submitted for publication.
25. Zobel, J., Derivation of Polymorphic Types for Prolog Programs, in: *Proceedings of the 4th International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 817-838.