Logic Programs as Types for Logic Programs*

Thom Frühwirth† ECRC
Ehud Shapiro‡ Weizmann Institute
Moshe Y. Vardi§ IBM Research
Eyal Yardeni¶ Weizmann Institute

Abstract

Type checking can be extremely useful to the program development process. Of particular interest are descriptive type systems, which let the programmer write programs without having to define or mention types. We consider here optimistic type systems for logic programs. In such systems types are conservative approximations to the success set of the program predicates. We propose the use of logic programs to describe types. We argue that this approach unifies the denotational and operational approaches to descriptive type systems and is simpler and more natural than previous approaches. We focus on the use of unary-predicate programs to describe types. We identify a proper class of unary-predicate programs and show that it is expressive enough to express several notions of types. We use an analogy with 2-way automata and a correspondence with alternating algorithms to obtain a complexity characterization of type inference and type checking. This characterization was facilitated by the use of logic programs to represent types.

1 Introduction

It has long been recognized that type-checking can be extremely useful to the program development process. Type checking enables automatic detection of many programming errors and it increases confidence in the correctness of programs. Furthermore, type information can be used also by the compiler for program optimization. Of particular interest are *descriptive* type systems, which let the programmer write programs without having to define or mention types; rather, the compiler automatically infers types and checks for type correctness [Red88].

Given that recognition in the benefit of type checking, the design of type systems for logic programming languages has been studied extensively (cf. [Klu87, Mis84, MO83, Red88, XW88, Zob87]). The basis for descriptive type systems for logic programs was proposed by Mishra [Mis84]: a formula that fails may be considered erroneous. Thus, the type of a predicate describes all the terms for which the predicate *may* succeed. Such types can be called *optimistic types* [Red88].

A type of a predicate in a logic program is therefore a conservative approximation to the meaning of that predicate, i.e., it must be a superset of the success set of the predicate. There can be, however, more than one such superset. In choosing a type, the issues that have to be considered involve the tightness of the approximation, its representation, and its computational complexity. Several proposals have been studied in the literature (see [HJ90b] for a survey). All of them share the following basic intuition, which originated in [Mis84] and made explicit in [Red90].

> First, let us consider a simplified situation. Let p be a binary predicate. We can consider an atom $P(t_1, t_2)$ to be meaningful if there is a term u_1 such that $P(u_1, t_2)$ is true, and symmetrically, there is a term u_2 such that $P(t_1, u_2)$ is true. The rationale is that if there is a u_1 such that $P(u_1, t_2)$ is true, then t_2 is a legitimate argument to p. However, if there is no such u_1, then t_2 is not covered in the second argument in any of the clauses for p. Hence, the atom is most likely erroneous and it is reasonable to interpret it

*Part of this work was done while the first and third authors were visiting the Weizmann Institute

†ECRC, Arabellastrasse 17, D-8000 Muenchen 81, Germany, email: thom@ecrc.de

‡Dept. of Applied Math., Weizmann Institute of Science, P.O.Box 26, 76 100 Rehovot, Israel, email: udi@wisdom.weizmann.ac.il

§IBM Almaden Research Center K53-802, 650 Harry Rd., San Jose, CA 95120-6099, USA, email: vardi@ibm.com

¶Dept. of Applied Math., Weizmann Institute of Science, P.O.Box 26, 76 100 Rehovot, Israel, email: eyal@wisdom.weizmann.ac.il

CH3025-4/91/0000/0300$01.00 © 1991 IEEE

as "meaningless".

Heintze and Jaffar [HJ90b] study the relationship between two of the major approaches in the literature to defining descriptive types for logic programs. The first approach can be thought of as denotational. It extracts from the program set-theoretical constraints for the types; these set-theoretical constraints are expressed in any of various ad-hoc formalisms (cf. [Mis84, HJ90a]). The type assignment is then a preferred solution of these constraints. The second approach can be thought of as operational. It starts with an approximation of the immediate consequence operator T_P associated with a program P. Types are then defined as the fixpoint of the approximate operators [YS87, YS89]. The main result in [HJ90b] is that various notions of type obtained via the denotational approach are equivalent to the various notions of type obtained via the operational approach. The main result in [HJ90a] is the use of the denotational approach to develop type inference and type checking algorithms. (Type inference is the extraction of a type description from the program. Type checking is the determination whether a given goal is well-typed.)

In this paper we propose a unification of the denotational and operational approaches. Basically, we advocate using logic programs to represent types, and, following [YS87], we emphasize the use of unary-predicate programs (unary-predicate programs contain only unary predicates symbols but may contain nonunary function symbols). Strictly speaking, unary-predicate programs cannot represent types of nonunary predicates; but, as the intuition quoted above shows, our real interest is in types of predicate arguments – predicate types are essentially the cross product of these simpler types. Since types of predicate arguments are simply sets of terms, we contend that unary-predicate logic programs ought to be adequate to represent these sets. The argument in favor of our position goes, however, deeper than that. As explained above, the denotational approach extracts the set-theoretical constraints from the program. Our approach does essentially the same but in a kinder and gentler way; it simply converts the original program into a program that expresses the types of predicate arguments in the original program. Alternatively, one can view our unary-predicate programs in an operational way, as definitions of approximate consequence operators. Instead, however, of expressing these approximate operators in some other formal language, we express them in the same manner that the original consequence operator T_P was expressed, by logic rules. We contend that this approach is simpler and more natural than previously studied approaches.

Beyond the conceptual argument in favor of our approach, we believe that it also offers practical advantages. First, types as logic programs are easier to understand. Being able to represent types in the same formalism of the original program, whether one prefers to think about types denotationally or operationally, greatly facilitates the exploration of different notions of types. Second, the representation of types as unary-predicate programs is conducive to studying a critical aspect of type systems, which is their computational complexity.

Computational complexity is the raison d'être of descriptive type systems for logic programs; the only reason for us to approximate the success set of a predicate is that the success set is typically undecidable. It is crucial, therefore, for our types to be decidable, and identifying the complexity of types is of paramount importance. Unfortunately, previous works on descriptive type systems for logic programs, including [HJ90a, HJ90b], did not address the issue of computational complexity. To address this issue, we identify a class of unary-predicate programs, which we call *proper* unary-predicate programs. This class of programs is defined by a certain syntactic restriction on the rules that limits their ability to manipulate terms. Nevertheless, it turns out that several notions of types, e.g., the types defined by T_P in [HJ90a], as well as the types defined by "path abstraction" in [YS89] and by "path projection" in [Frü89], can be represented by proper unary-predicate programs. We use an analogy between proper unary-predicate logic programs and 2-way automata as well as the natural correspondence between logic programs and *alternating* algorithms (cf. [Sha84]), to study the complexity of type inference and type checking for types described by proper unary-predicate programs. The restriction imposed on such programs enables us to use *unfolding*[1] techniques, inspired by classical techniques in the theory of 2-way automata, to reduce proper unary-predicate programs to *regular programs* — these are programs that define regular sets of terms in the automata-theoretic sense.[2] This transformation can be viewed as type inference. Using alternating algorithms we then provide a precise characterization for the complexity of type checking.

[1] Unfolding is a basic technique of doing compile-time derivations in order to eliminate runtime derivations. It is used in partial evaluation, program transformation, program analysis, and program specialization; cf. [TS84, Eea88].

[2] A set of terms is regular if it is definable by a finite tree automaton [Tha73].

We believe that it is our use of logic programs to represent types that facilitated this characterization.

2 Preliminaries

We refer the reader to [Llo87] for standard terminology and definitions about logic programs. The denotational semantics of a logic program defines the *success set* of a program P as the minimal model of P viewed as a universal Horn theory. The operational semantics of P is defined in terms of the immediate consequence operator T_P associated with P. T_P operates on Herbrand interpretations; its definition is

$$T_P(\mathcal{I}) = \{H\theta \mid H \leftarrow \mathbf{B} \in P \text{ and } \mathbf{B}\theta \in \mathcal{I}\},$$

where θ ranges over ground substitutions. It is known that the success set of P is precisely the least fixpoint of T_P.

As observed in [HJ90b], the operational definition consists of three main components: (i) the collection of unifiers corresponding to the body atoms of a rule, (ii) the applications of these unifiers to the head of a rule, and (iii) the joining of the resulting sets, one from each rule head. We can approximate the success set of P by approximating T_P. To obtain such an approximation, we replace the above components of T_P by approximate ones.

We first describe in detail the approximation $\mathcal{T}_P$ defined in [HJ90a]; later on we onsider another approximation. The main feature of the $\mathcal{T}_P$ approximation is the replacement of substitutions by *set substitutions*, i.e., substitutions that map variables to sets of terms. Set substitution can be naturally viewed as a mappings from terms to sets of ground terms: if t is a term with variable occurrences $X_1, \ldots, X_k$ (note that we distinguish between multiple occurrences of the same variable) and α is a set substitution, then

$$t\alpha = \{t(X_1/t_1, \ldots, X_k/t_k) \mid t_i \in \alpha(X_i), 1 \leq i \leq k\}.$$

If Θ is a collection of ground substitutions over a set $\mathbf{X}$ of variables, then $A_{\mathbf{X}}(\Theta)$ is the set substitution over $\mathbf{X}$ that maps each variable $X \in \mathbf{X}$ to the set $\{X\theta \mid \theta \in \Theta\}$. The approximate consequence operator $\mathcal{T}_P$ is defined in [HJ90a] by:

$$\mathcal{T}_P(\mathcal{I}) = \{a \in H\alpha \mid H \leftarrow \mathbf{B} \in P \text{ and } \alpha = A_{var(H)}(\{\theta \mid \mathbf{B}\theta \in \mathcal{I}\})\},$$

where θ ranges over ground substitutions and $var(H)$ is the set of variables in the head H. As explained in [HJ90a], $\mathcal{T}_P$ first collects together the ground substitutions for a rule that instantiate the body atoms into elements of $\mathcal{I}$. ¿From these substitutions it collects all the possible values that each variable may be instantiated to, ignoring the relationship between these variables. A set substitution is then defined as the mapping from each variable into the collected set of values, and finally, this set substitution is applied to the head of the rule. Since the success set of P is defined as the least fixpoint T_P, and $\mathcal{T}_P$ approximates T_P, the least fixpoint of $\mathcal{T}_P$ is a conservative approximation of the success set of P.

3 Type Programs

As observed in Section 2, $\mathcal{T}_P$ collects all the possible values that each variable may be instantiated to, ignoring the relationship between these variables. The idea underlying our approach is that the operator $\mathcal{T}_P$ should be expressed in the same way that the original operator T_P was expressed – by logic rules.

To this end we rewrite a program P into a program $P_{\mathcal{T}}$, such that the success set of $P_{\mathcal{T}}$ is essentially the least fixpoint of $\mathcal{T}_P$. We call $P_{\mathcal{T}}$ a *type program*. We can assume without loss of generality that no two rules in P have a variable in common. With each variable X in P we associate a unary predicate x. We refer to these predicates as the *unary predicates*. Intuitively, the success set of the predicate x approximates the set of instantiations of the variable X. We also introduce a new unary predicate *type*. Finally, with each k-ary predicate symbol p in P we associate a k-ary function symbol f_p. Intuitively, $f_p(\mathbf{t})$ will be in the success set of *type* precisely when $\mathbf{t}$ is in the type of p.

Let

$$p(\mathbf{t}) \text{:-} p_1(\mathbf{t}_1), \ldots, p_m(\mathbf{t}_m)$$

be a rule in P, with head variables $X_1, \ldots, X_k$. Since $\mathcal{T}_P$ decouples multiple occurrences of variables in the head, we associate a distinct variable X_j^i with the ith occurrence of X_j in the head. Let $\tilde{\mathbf{t}}$ be the "decoupled" version of $\mathbf{t}$. That is, $\tilde{\mathbf{t}}$ is obtained from $\mathbf{t}$ be replacing the ith occurrence of X_j by X_j^i. Note that $\tilde{\mathbf{t}}$ does not have *repeated variables*, i.e., multiple occurrences of the same variables, even though $\mathbf{t}$ may have repeated variables. For the above rule we put several rules in $P_{\mathcal{T}}$. We first put one rule for the head:

$$type(f_p(\tilde{\mathbf{t}})) \text{:-} x_1(X_1^1), \ldots, x_k(X_k^l).$$

The body of this rule contains a literal $x_j(X_j^i)$ for the ith occurrence of X_j in the head of the original rule. We now add a rule for each variable X_i:

$$x_i(X_i) \text{:-} type(f_{p_1}(\mathbf{t}_1)), \ldots, type(f_{p_m}(\mathbf{t}_m)).$$

If the head $p(\mathbf{t})$ contains no variable, then we pretend that it contains a new variable X. Thus, we put in $P_{\mathcal{T}}$ one rule for the head:

$$type(f_p(\tilde{\mathbf{t}})) \text{:-} x(X),$$

and one rule for the body

$$x(X) \text{:-} type(f_{p_1}(\mathbf{t}_1)), \ldots, type(f_{p_m}(\mathbf{t}_m)).$$

(Thus, if the body fails, then the predicate x is empty, which forces the head to fail.)

The basic idea of this construction is to define the type of the head predicate in terms of the types of the head variables and to define the types of the head variables in terms of the types of the body predicates. Furthermore, each occurrence of head variables is typed independently, so multiple occurrences of the same variable are decoupled.

Example 3.1: Let P be the program for list reversal:

$$\begin{array}{lll} rev(nil, U, U) & :\text{-}. & \\ rev(X.L_1, L_2, L_3) & :\text{-} & rev(L_1, X.L_2, L_3). \end{array}$$

Then $P_{\mathcal{T}}$ is the program

$$\begin{array}{lll} type(f_{rev}(nil, U_1, U_2)) & :\text{-} & u(U_1), u(U_2). \\ type(f_{rev}(X.L_1, L_2, L_3)) & :\text{-} & x(X), l_1(L_1), \\ & & l_2(L_2), l_3(L_3). \end{array}$$

$$\begin{array}{lll} u(U) & :\text{-}. & \\ x(X) & :\text{-} & type(f_{rev}(L_1, X.L_2, L_3)). \\ l_1(L_1) & :\text{-} & type(f_{rev}(L_1, X.L_2, L_3)). \\ l_2(L_2) & :\text{-} & type(f_{rev}(L_1, X.L_2, L_3)). \\ l_3(L_3) & :\text{-} & type(f_{rev}(L_1, X.L_2, L_3)). \end{array}$$

∎

The following theorem asserts that $P_{\mathcal{T}}$ indeed approximate P in the desired manner. The proof is actually quite simple; the claim follows almost immediately from the close correspondence between the program $P_{\mathcal{T}}$ and the definition of $\mathcal{T}_P$. Essentially, $\mathcal{T}_P$ can be viewed as the immediate consequence operator $T_{P_{\mathcal{T}}}$ of $P_{\mathcal{T}}$.

Theorem 3.2: *Let P be a program, let $P_{\mathcal{T}}$ be its associated type program, and let p be a predicate of P. Then $type(f_p(\mathbf{t}))$ is in the success set of $P_{\mathcal{T}}$ precisely when $p(\mathbf{t})$ is in the least fixpoint of $\mathcal{T}_P$.*

To appreciate the simplicity of our approach we urge the reader to compare our approach with the rather involved set equations of [HJ90a]. We contend that the natural way to express optimistic type is by means of type programs. The two approaches of approximation that are studied in [HJ90b], the denotational and the operational, simply coincide with the two ways of defining the semantics of type programs, denotationally or operationally. Our approach, however, offers more than just a unifying notation, as we shall now see.

The transformation of P into $P_{\mathcal{T}}$ yields a representation of types by unary-predicate type programs. Unary-predicate programs, however, can still describe very complicated types. It is easy to see that unary-predicate programs can simulate arbitrary programs, by converting tuples of terms to single terms (as we did with the *type* predicate). Thus, it is not clear that the use of unary-predicate type programs offers any benefit. Fortunately, the type programs $P_{\mathcal{T}}$ satisfy a certain syntactical restriction that we will be able to utilize.

Let t and t' be terms. We say that t and t' are ***disjoint*** if they have no variable in common. If t' is a subterm of t, then we say that t' is a ***strong subterm of t*** (or, equivalently, t is a ***strong superterm*** of t') if whenever x is a variable of t' then it occurs in t only as a variable of t'. Intuitively, if we think in terms of the directed acyclic graph representing t, then there is no path from t to a variable x of t' that does not go through t'.

It is easy to see that type programs satisfy the following property.

Proposition 3.3: *Let $p(t)$:-$p_1(t_1), \ldots, p_k(t_k)$ be a rule in $P_{\mathcal{T}}$. Then t does not contain repeated variables, and each t_i is either a subterm of t, a strong superterm of t, or disjoint from t.*

Intuitively, rules in $P_{\mathcal{T}}$ are limited in their ability to manipulate terms. We call unary-predicate programs that obey the restrictions in Proposition 3.3 *proper*. Proper unary predicate programs cannot have rules such as

$$p(g(f(X), Y)) \text{:-} q(g(X, f(Y))).$$

With such rules one can easily simulate Turing machines [Sha84]. Thus, improper unary-predicate programs can define all recursively enumerable sets of terms. In contrast, as we shall see, proper unary-predicate programs define regular sets of terms. Such sets are often used to describe types, cf. [HJ90a,

Mis84, YS87, Zob87], though often only a proper subclass of the class of regular term sets is used.

4 Type Inference and Type Checking

Our goal in this section is to characterize the complexity of type inference and type checking for types described by unary-predicate programs. As a first step we convert proper unary-predicate programs to *uniform* unary-predicate programs. A unary-predicate program is uniform if whenever $p(t)$:-$p_1(t_1),\ldots,p_k(t_k)$ is a rule in the program then t does not contain repeated variables and either each t_i is a subterm of t, each t_i is a strong superterm of t, or each t_i is disjoint from t. It is not hard to see that by introducing auxiliary unary-predicates we can transform proper unary-predicate programs to uniform unary-predicate programs; the auxiliary predicates are used to separate subterm and superterm goals. Note that the type programs generated in the previous section are uniform.

Proposition 4.1: *If P is a proper unary-predicate program, then there exists a uniform unary-predicate P' such that the projection of the success set of P' on the predicates of P is the success set of P.*

If all the terms in the body of a rule are strong superterms of the head term, then we call the rule *depth increasing.* The second step is to convert uniform unary-regular programs to *regular* unary-predicate programs by eliminating depth-increasing rules. A unary-predicate program is regular if whenever $p(t)$:-$p_1(t_1),\ldots,p_k(t_k)$ is a rule in the program then t does not contain repeated variables and each t_i is a subterm of t or disjoint from t. The transformation of uniform unary-predicate programs to regular unary-predicate programs is quite involved and is the technical crux of our results.

As we show, there is a natural algorithm for determining membership in the success set of regular unary-predicate programs, which means that regular unary-predicate programs are amenable to *type checking.* Thus, the transformation of uniform unary-predicate programs to regular unary-predicate programs can be viewed as *type inference.* Again, we urge the reader to compare our approach to type inference and type checking with that of [HJ90a] to appreciate its conceptual and algorithmic simplicity.

In what follows we discuss first regular unary-predicate programs, and then show that uniform programs can be converted to regular programs.

4.1 Regular Programs

We focus here on type checking for regular unary-predicate programs – determining whether a given goal $p(t)$ is well-typed, i.e., whether $p(t)$ is in the success set of a given regular unary-predicate program P (note that P need not be a type program). Consider a derivation tree of the program P for a goal $p(t)$.[3] All the terms that occur in the tree are either subterms of t or variants of terms from P (variants are obtained by variable renaming). We call such trees *nonincreasing.* If there is no variable sharing between nodes of the tree, then the problem would have been easy, and existence of such a tree can be determined in polynomial time. Variable sharing among nodes in the tree complicates things considerably.

Theorem 4.2 : *Determining membership in the success set of regular unary-predicate programs is EXPTIME-complete.*

Sketch of Proof: To obtain the exponential time upper bound, we use the natural correspondence between logic programs and *alternating* algorithms (cf. [Sha84]). An alternating algorithm ([CKS81]) can branch both existentially and universally. The requirement is that at least one branch succeeds at a existential branching point and all branches succeed at a universal branching point.

Without loss of generality we assume that the rules are either of the form

$$p(f(X_1,\ldots,X_k)) \;:-\; p_1(X_1),\ldots,p_k(X_k),$$

$$p(X) \;:-\; p_1(X),\ldots,p_k(X),$$

$$p(X) \;:-\; p_1(t_1),\ldots,p_k(t_k),$$
where X is disjoint from the t_i's,
or of the form

$$p(c) \;:-\; p_1(t_1),\ldots,p_k(t_k).$$

We describe an alternating algorithm that tests whether a goal $p(t)$ is in the success set of a regular unary-predicate program P. The algorithm always has a set of goals under consideration, initially the set

[3] A derivation tree of the program P for a goal $p(t)$ is a tree labeled by atoms. the root is labeled by $p(t)$. If an internal node x is labeled by the atom $q(s)$, then there is in P a rule $q(s')$:-$q_1(s_1),\ldots,q_k(s_k)$ and there is a substitution θ such that $s = s'\theta$ and the children of x are labeled by the atoms $q_1(s_1)\theta,\ldots,q_k(s_k)\theta$.

consists of $p(t)$. The algorithm tries to prove that an instance of the set of goals under consideration is contained in the success set of P. The algorithm partitions the set of goals into the finest partition such that goals in different blocks do not share variables. Then the algorithm branches universally – each branch considers one block of the partition. For each goal $q(s)$, the algorithm now existentially selects a rule r from P, unifies the head of r with $q(s)$, and replaces $q(s)$ by the instantiated body of r.

It can be shown that the number of goals in a single branch is at most polynomial in the size of the program. It was shown in [CKS81] that an alternating polynomial-space algorithm can be simulated by an exponential-time algorithm. The upper bound follows.

To prove the exponential lower bound we first prove that the intersection problem for tree automata is EXPTIME-complete. This proof also uses alternation; it is known that exponential-time algorithms can be simulated by alternating polynomial-space algorithms [CKS81]. An alternating polynomial-space Turing machine accepts an input if it has an accepting computation tree. Given a machine M and an input of length n, we construct n tree automata of size $O(n)$ such that they have a nonempty intersection iff M has an accepting computation tree on the input. Finally, we show that the intersection problem can be reduced to the membership problem. ▮

It should be noted that the lower bound of Theorem 4.2 holds even for regular unary programs (i.e., where both predicate and function symbols are unary). Since for unary programs the approximation $\mathcal{T}_P$ is in fact precise, it follows that Theorem 4.2 implies an exponential-time lower bound on the complexity of type checking for this approximation.

A regular unary-predicate program is said to be *reduced* if it does not contain rules with nonempty bodies of the form:

$$p(X) \;:\!-\; p_1(t_1), \ldots, p_k(t_k),$$
where X is disjoint from the t_i's, or

$$p(c) \;:\!-\; p_1(t_1), \ldots, p_k(t_k).$$

In such rules the body is either true or false. If it is true then we can eliminate the body, and if it is false we can eliminate the rule. An algorithm similar to the algorithm described in the proof of Theorem 4.2 can be used to determine if the body is true or false.

Proposition 4.3: *There is an exponential-time algorithm that converts a regular unary-predicate program to a reduced program.*

The transformation of regular unary-predicate programs to reduced programs will come handy later on. We note that reduced regular unary-predicate programs can be viewed as alternating tree automata [Slu85]; such automata are known to be equivalent in expressive power (though not in succinctness) to standard tree automata. Thus, the success set of a regular unary-predicate program is indeed regular, which explains our terminology. The representation of types as tree automata let us answer many questions about types in addition to the membership question addressed in Theorem 4.2; we can test finiteness of types (cf. [Don65]), containment of types (cf. [Sei90]), etc..

4.2 From Uniform Programs to Regular Programs

As we saw in Section 4.1, there is a natural type-checking algorithm for regular unary-predicate programs. Consequently, our goal for type inference is the extraction of regular unary-predicate type programs. To understand the intuition of the transformation from uniform to regular unary-predicate programs, consider a derivation tree of a uniform unary-predicate program P for a goal $p(t)$. As we observed earlier, if the program is regular, then the depths of the intermediate terms is bounded. If the program is not regular, then the depths of the terms in the intermediate goals may increase and decrease and are potentially unbounded. One can view regular unary-predicate programs as analogous to 1-way automata, while uniform unary-predicate programs are analogous to 2-way automata. The critical observation is that the terms in the leaves of the derivation tree have bounded depth. Thus, whenever the depth of the terms in the subgoal increases, it must eventually decrease. By adding new rules we can eliminate increasing trees. This transformation is analogous to the transformation of 2-way automata into 1-way automata [RS59, She59]. We now sketch this transformation in more detail.

Without loss of generality we assume that the rules are either of the form

$$p(X) \;:\!-\; p_1(t_1), \ldots, p_k(t_k),$$

$$p(c) \;:\!-\; p_1(t_1), \ldots, p_k(t_k), \text{ or}$$

$$p(f(X_1, \ldots, X_k)) \;:\!-\; p_1(X_1), \ldots, p_k(X_k),$$

Furthermore, we can assume that the only rules with an empty body are of the form $p(c)$:–.

Consider now a derivation tree of a ground literal; this tree may be increasing. If x is a node of the tree labeled by a term t, then the children of x are labeled either by immediate subterms of t, by strong superterms of t, or by variant terms of P disjoint from t. The problematic case is where x is an *increasing node*, i.e., the children terms are proper superterms of the parent term. Suppose indeed that x is an increasing node. We know, however, that the leaves of the tree are labeled by terms of depth 0. Consider a path from x towards a leaf. The terms on this path ought to be strong superterms of t until we either reach a node labeled by a variant term from P disjoint from t or we reach a node labeled again by t. Thus, there is a finite subtree underneath x whose frontier is labeled either by variant terms from P disjoint from t or by t. We call this subtree a *stationary subtree*, since, as far as its frontier is considered, it neither constructs superterms of t nor does it decompose t into its subterms.

A stationary subtree corresponds to a nonground incomplete (i.e., with some unexpanded leaves) derivation tree whose root is labeled by X, whose internal nodes are labeled by terms properly containing X, and whose leaves are labeled by variant terms from P disjoint from X or by X. We call such a tree a *stationary* tree. This tree corresponds to a rule of the form

$$p(X) \text{:–} p_1(t_1), \ldots, p_k(t_k),$$

where each t_i is either a variant term from P disjoint from X or is X. This rule can be obtained from P by an unfolding that starts with an increasing rule. We call such rules *stationary* rules, since they neither construct superterms on top of the head term nor do they decompose the head term into its subterms. Clearly, the number of body literals of the form $p_i(X)$ is at most linear in the size of the program. Thus, to get a bound on the size of stationary rules we need to bound the number of body literals with variant terms from P. A careful analysis shows that if $p_{i_1}(t_{i_1}), \ldots, p_{i_l}(t_{i_l})$ are literals in the body of a stationary rule and $t_{i_1}, \ldots, t_{i_l}$ have a shared variable, then $t_{i_1}, \ldots, t_{i_l}$ are variant terms from a single rule in P. Thus, the size of a stationary rule is at most exponential in the size of P. It follows that the number of possible stationary rules is at most doubly exponential in the size of P.

Since the stationary rules obtained by unfolding do follow from the rules of P, we can add them to the program without changing its semantics. The advantage of adding the stationary rules to the program is that they correspond to stationary subtrees in an increasing derivation tree. If a stationary rule r that corresponds to a stationary subtree τ at a node x has been added to P, then we can eliminate τ from the derivation tree by deleting all the internal nodes of τ and connecting the leaves of τ to x. Note that after this operation, x is not an increasing node anymore. Thus, if P is augmented with all its stationary rules, then there is no need to use its increasing rules any more and they can be eliminated. But if all the increasing rules are eliminated, then we are left with a regular program.

We have essentially shown the following:

Theorem 4.4: *Let P be a uniform unary-predicate program. Then there is a regular unary-predicate program P' that is equivalent to P.*

Example 4.5: The program $P_{\mathcal{T}}$ of Example 3.1 is equivalent to the regular program

$$\begin{array}{lll} type(f_{rev}(nil, U_1, U_2)) & \text{:–}. & \\ type(f_{rev}(X.L_1, L_2, L_3)) & \text{:–} & l_1(L_1). \\ u(U) & \text{:–}. & \\ x(X) & \text{:–}. & \\ l_1(nil) & \text{:–}. & \\ l_1(X.L_1) & \text{:–} & l_1(L_1). \\ l_2(L_2) & \text{:–}. & \\ l_3(L_3) & \text{:–}. & \end{array}$$

∎

Theorem 4.4 tells us that uniform unary-predicate programs define regular sets of terms, but it does not tell us how to find the stationary rules that need to be added to the program. We know that these rules can be obtained by unfolding, but because of the increasing rules in the program, there is no a priori bound on the size of the necessary unfolding. We now prove such a bound.

Let P be a uniform program. The *depth* of P is the maximum depth of a term in an increasing rule of P. Let α be a stationary tree. The *depth* of α is the maximum depth of a term in α containing the root variable X.

Proposition 4.6: *Let P be a uniform unary-predicate program, and let r be a stationary rule of P. Then r can be obtained from a stationary tree whose depth is at most doubly exponential in the size of P.*

Sketch of Proof: Let d be the depth of P. Let R_i be the set of all stationary rules that can be obtained

from stationary trees of depth at most i. Clearly $R_i \subseteq R_{i+1}$ for all $i \geq 1$. We claim that if $R_i = R_{i+d}$, then $R_i = R_j$ for all $j \geq i + d$. Suppose that $R_i = R_{i+d}$. Consider now a stationary rule that is obtained from a stationary tree α of depth $i+d+1$. That is, there is a node x in the tree labeled by a term $t(X)$ of depth $i+d+1$. Consider the path from the root to x. There must be a node y on that path that is labeled with a term $t'(X)$ of depth at most d such that all nodes between y and x are labeled with superterms of $t'(X)$. Thus, y is a root of subtree whose internal nodes are labeled by superterms of $t'(X)$ and whose leaves are labeled by variant terms of P disjoint from $t'(X)$ or by $t'(X)$. If we replace $t'(X)$ by X, then this subtree is actually a stationary tree of depth between $i+1$ and $i+d$. Since we assumed that $R_i = R_{i+d}$, we can replace this stationary tree by a tree of depth at most i. The maximum term depth in the new subtree is now $i+d$. By repeating this process we can replace α by a stationary tree of depth at most $i+d$.

Let m be the number of stationary rules; as we observed earlier, m is at most doubly exponential in the size of the given program. We showed that if $R_i = R_{i+d}$, then $R_i = R_j$ for all $j \geq i + d$. Thus, the sequence $R_1, R_2, \cdots$ must converge in at most dm steps, i.e., $R_j = R_{dm}$ for all $j \geq dm$. ∎

Proposition 4.6 provides a doubly-exponential time upper bound on the conversion of uniform unary-predicate programs to regular unary-predicate programs. Combining this with the bound of Theorem 4.2, we get a triply-exponential time upper bound for determining membership in the success set of proper unary-predicate programs. It turns out that using the ideas of Proposition 4.3 we can provide exponential upper bounds for both type inference and type checking.

Theorem 4.7:

1. *There is an exponential-time algorithm that converts proper unary-predicate programs to equivalent reduced regular unary-predicate programs.*
2. *Determining membership in the success set of proper unary-predicate programs can be done in exponential time.*

We note that decidability for unary programs, i.e., programs where both the predicates and the functions are unary, was given as a corollary of the main results in [AH84]. That result, however, follows easily from Rabin's decidability result for $S\omega S$ [Rab69], since the set of unary terms can be viewed as an infinite tree. The main difficulty in our work is in dealing with nonunary functions. It may seem that result of Theorem 4.7 is orthogonal to the decidability result of [AH84], since we require that the program be proper. It is not, however, hard to see that all unary programs can be expressed as proper unary programs by adding some auxiliary predicates. Thus, Theores 4.2 and 4.7 provide precise bounds for the complexity of unary programs.

5 Types by Paths

So far we focused on the approximation $\mathcal{T}_P$. Our approach, however, applies to other approximations as well. In this section we introduce another approximation, which is looser than $\mathcal{T}_P$. We show that this approximation can also be described by proper unary-predicate programs.

The essence of the approximation $\mathcal{T}_P$ is the decoupling between types of variables. The approximation we describe now goes much further; it completely decouples *paths*.

Intuitively, a path is a branch in the parse tree of a term or an atom. For example, consider the term $f(g(a,a),b)$; its paths are the strings $f_1(g_1(a))$, $f_1(g_2(a))$, and $f_2(b)$. Formally, the set of paths of a term (atom) t is a set of unary terms (atoms). These terms are built from the constant symbols and variables in t by means of new unary function and predicate symbols: for each k-ary function (predicate) symbol f (p) we introduce unary function (predicate) symbols $f_1, \ldots, f_k$ ($p_1, \ldots, p_k$). The set of paths of a term or an atom t, denoted $paths(t)$, is defined as follows, where c denotes a constant symbol, X denotes a variable, and h denotes a function or predicate symbol:

- $paths(c)$ is the set $\{c\}$,
- $paths(X)$ is the set $\{X\}$,
- $paths(h(t_1, \ldots, t_k))$ is the set

$$\{h_i(t') \mid t' \in paths(t_i), 1 \leq i \leq k\}.$$

For a set T of terms, we define $paths(T) = \cup_{t \in T} paths(t)$.

The path approximation operator π_P is defined in [YS89] by:

$$\pi_P(\mathcal{I}) = \{a \mid H \leftarrow \mathbf{B} \in P, \\ paths(a) \subseteq paths(H\theta), \text{ and} \\ paths(\mathbf{B}\theta) \subseteq paths(\mathcal{I})\},$$

where θ ranges over ground substitutions. Intuitively, *paths* breaks all terms and atoms into their constituents paths, and π_P operates over the "path base" rather than over the Herbrand base.

We now describe the type program P_π associated with a program P that corresponds to the approximation π_P. Let

$$p(\mathbf{t}) \text{:-} p_1(\mathbf{t}_1), \ldots, p_m(\mathbf{t}_m)$$

be a rule in P. For each atom a in $paths(p(\mathbf{t}))$ we put a rule

$$a \text{:-} paths(p_1(\mathbf{t}_1)), \ldots, paths(p_m(\mathbf{t}_m)).$$

These rules are called *path rules*. We then add one rule for the head:

$$p(\mathbf{t}) \text{:-} paths(p(\mathbf{t})).$$

This rule is called a *head rule*.

Example 5.1: Let P be the program:

$$\begin{array}{ll} p(f(X,Y)) & \text{:-} q(X), r(Y) \\ p(f(U,V)) & \text{:-} s(U), t(V) \\ q(a) & \text{:-} \\ r(b) & \text{:-} \\ s(c) & \text{:-} \\ t(d) & \text{:-} \end{array}$$

Then $\pi_{\mathcal{T}}$ is the program

$$\begin{array}{ll} p_1(f_1(X)) & \text{:-} q_1(X), r_1(Y) \\ p_1(f_2(Y)) & \text{:-} q_1(X), r_1(Y) \\ p_1(f_1(U)) & \text{:-} s_1(U), t_1(V) \\ p_1(f_2(V)) & \text{:-} s_1(U), t_1(V) \\ q_1(a) & \text{:-} \\ r_1(b) & \text{:-} \\ s_1(c) & \text{:-} \\ t_1(d) & \text{:-} \\ p(f(X,Y)) & \text{:-} p_1(f_1(X)), p_1(f_2(Y)) \\ p(f(U,V)) & \text{:-} p_1(f_1(U)), p_1(f_2(V)) \\ q(a) & \text{:-} q_1(a) \\ r(b) & \text{:-} r_1(b) \\ s(c) & \text{:-} s_1(c) \\ t(d) & \text{:-} t_1(d) \end{array}$$

∎

We leave it to the reader to verify that P_π indeed approximate P in the desired manner. The observant reader probably noticed that P_π is not unary and is not proper. Note, however, that P_π consists of two parts: the part consisting of the path rules and the part consisting of the head rules, where the former does not depend at all on the latter. Furthermore, the part consisting of the path rules is unary (both predicate and function symbols are unary), and therefore, as observed in Section 4, can be rewritten as proper. It follows that the success set of P_π is decidable. Note that the vocabulary of P_π is larger than the vocabulary of P; nevertheless, P_π can be transformed to an equivalent program with the same vocabulary as P. While $\mathcal{T}_P$ approximates the sucess set of P better π_P, practical experience has shown that π_P is often an adequate approximation [YS89]. It remains to be seen whether in practice the loosening of the approximation yields any reduction in the complexity of type inference and type checking.

6 Concluding Remarks

We argued that the natural way to express types of logic programs is by logic programs. We showed that this approach unifies the denotational and operational approaches to descriptive types. We identified a class of proper unary-predicate program, showed their adequacy for type description, and characterized the complexity of type inference and type checking for such type programs. Our complexity bounds are obtained by means of an unfolding technique inspired by classical techniques in the theory of 2-way automata and by the use of alternating algorithms. Our exponential-time lower bound on the complexity of type checking for the $\mathcal{T}_P$ approximation raises the intriguing research questions of the practical complexity of our algorithms and of finding descriptive types with polynomial-time algorithms for type inference and type checking.

References

[AH84] D. Angluin and D. N. Hoover. Regular prefix relations. *Mathematical System Theory*, 17:167–191, 1984.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28:114–133, 1981.

[Don65] J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:819, 1965.

[Eea88] A. P. Ershov et al., editor. *Special Issue on Partial Evaluation and Mixed Computation.* New Generation Computing 6:2-3, 1988.

[Frü89] T.W. Frühwirth. Type inference by program transformation and partial evaluation. In Abramson H. and M. H. Rogers, editors,

Meta-Programming in Logic Programming. MIT Press, 1989.

[HJ90a] N. C. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 197–209, 1990.

[HJ90b] N. C. Heintze and J. Jaffar. Semantic types for logic programs. unpublished manuscript, 1990.

[Klu87] F. Kluzniak. Type synthesis for ground Prolog. In *Proc. 4th International Conference on Logic Programming*, pages 788–816, Melbourne, Australia, May 1987. MIT Press.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[MO83] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. In *Logic Programming Workshop*, pages 107–121, 1983.

[Rab69] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. AMS*, 141:1–35, 1969.

[Red88] U.S. Reddy. Notions of polymorphism for predicate logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th International Conf. and Symp. on Logic Programming*, Seattle, Washington, 1988. MIT Press.

[Red90] U.S. Reddy. A perspective on types for logic programs. unpublished manuscript, 1990.

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3:114–125, 1959.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19:424–437, 1990.

[Sha84] E. Shapiro. Alternation and the computational complexity of logic programs. *J. Logic Programming*, 1:19–34, 1984.

[She59] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM J. Res. Dev.*, 3:199–201, 1959.

[Slu85] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[Tha73] J.W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, chapter 4, pages 143–172. Prentice-Hall, 1973.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. of 2nd Int'l Logic Programming Conference*, pages 127–138, Uppsalla, 1984.

[XW88] J. Xu and D. S. Warren. A type inference system for Prolog. In *Proc. 5th International Conf. and symp. on Logic Programming*, pages 604–619, Seattle, Washington, August 1988.

[YS87] E. Yardeni and E. Shapiro. A type system for logic programs. In Ehud Shapiro, editor, *Concurrent Prolog*, chapter 28. MIT Press, 1987.

[YS89] E. Yardeni and E. Shapiro. A type system for logic programs. Technical Report CS89-03, The Weizmann Institute of Science, 1989.

[Zob87] J. Zobel. Derivation of polymorphic types for Prolog programs. In *Proc. 4th International Conference on Logic Programming*, pages 817–838, Melbourne, Australia, May 1987. MIT Press.