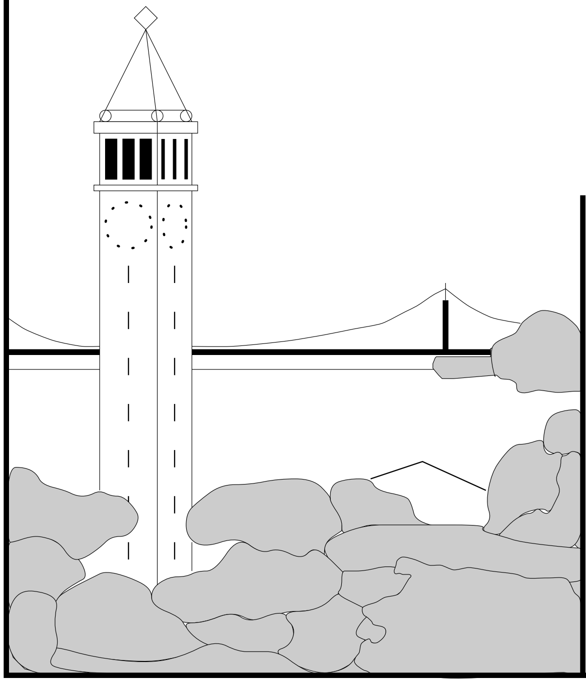
Directional Type Checking of Logic Programs

Alexander Aiken
Computer Science Division
University of California, Berkeley
571 Evans
Berkeley, CA 94720
email: aiken@cs.berkeley.edu

T. K. Lakshman
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave,
Urbana, IL 61801
email: lakshman@cs.uiuc.edu



Report No. UCB/CSD 94-791

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

We present an algorithm for automatic type checking of logic programs with respect to *directional types* that describe both the structure of terms and the directionality of predicates. The type checking problem is reduced to a decidable problem on systems of inclusion constraints over set expressions. We discuss some properties of the reduction algorithm, complexity, and present a proof of correctness.

1 Introduction

Most logic programming languages are untyped. In Prolog, for example, it is considered meaningful to apply any n-ary predicate to any n-tuple of terms. However, it is generally accepted that static type checking has great advantages in detecting programming errors early and for generating efficient executable code. Motivated at least in part by the success of type systems for procedural and functional languages, there is currently considerable interest in finding appropriate definitions of *type* and *well-typing* for logic languages. This paper explores the type checking problem for *directional types*, a recently proposed, and very rich, idea for types that describe both the structure of terms and the directionality of predicates.

Most type systems for logic programming languages define a type as a set of ground terms and adopt the view that the purpose of type analysis is to compute an approximation to the success set of a program; i.e., to describe the set of terms for which a predicate is true [Mis84, MR85, HJ90a, HJ92, DZ92]. While knowing something about the success set is useful, it lacks some basic properties expected of type systems. In particular, knowing only the success set does not help in reasoning accurately about the relationship between program inputs (initial goals) and program outputs (resolved goals). To do this requires reasoning about the *directionality* of predicates. Procedural and functional languages are *directional*: some distinguished values are designated as input; other distinguished values are computed as output. In contrast, logic programming is *non-directional*: conceptually, one can execute a logic program that defines a predicate by specifying any subset of the predicate's arguments as input; the remaining arguments are computed as output. In practice most logic programs are directional since predicate definitions are often used only in one direction or at most a few directions [Deb89]. Further, directionality greatly simplifies reasoning about termination and complexity properties of programs. Consequently, so-called *mode* systems for logic programs have been developed to capture directionality of predicates. Most mode systems distinguish between input and output arguments to predicates or between ground arguments (these are considered input) and non-ground arguments (these are considered output). However, the input/output or ground/non-ground distinctions made by most mode systems do not permit reasoning about the structure of terms.

Recently, Bronsard et. al. [BLR92] have proposed a combination of modes and types that we call *directional types*. Directional types specify both the directionality of predicates and the structure of the arguments to the predicate. The use of directional types by Apt and others for reasoning about partial correctness [Apt93],and for use in compiler optimizations [AE93], suggests that a uniform view of types and directionality is indeed useful. None of these previous works has addressed the type checking problem for logic programs with directional types.

It is worthwhile to explain our introduction of the term *directional type*. What we call a directional type is called a *mode dependence* in [BLR92] and a *type* in [Apt93]. Besides the problems caused by having two terms for the same concept, there is the added complication that both terms have multiple conflicting definitions in the literature. To avoid confusion over terminology, we prefer to introduce a fresh and hopefully descriptive name.[1]

[1]The name "directional type" has also been adopted by Bronsard et. al. in their subsequent work [BLR93] on a polymorphic type system and used for proving termination of logic programs with incomplete data structures.

$$\texttt{append(nil,Y,Y).}$$
$$\texttt{append(W.X,Y,W.Z)} \leftarrow \texttt{append(X,Y,Z).}$$

(a) The append predicate.

$$(\mathit{List}(1), \mathit{List}(1), 1) \rightarrow (\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))$$
$$(1, 1, \mathit{List}(1)) \rightarrow (\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))$$

(b) Directional types for appending and splitting lists.

Figure 1: A sample program and directional types.

Intuitively, a *type* is a set of terms T, and a term t *has type* T if $t \in T$.[2] For example, let "1" be the type denoting the set of all terms and let $\mathit{List}(T)$ be the type denoting the set of all lists with elements drawn from T; i.e., $\mathit{List}(T) = \{\texttt{nil}, t_1.\texttt{nil}, t_2.t_1.\texttt{nil}, \ldots\}$ where $t_i \in T$. A *directional type* is an implication $A \rightarrow B$, where A and B are types. Semantically, a directional type $A \rightarrow B$ is an assertion that if a goal has type A and the program terminates successfully, then the result has type B. Two directional types of the `append` predicate are given in Figure 1b. The first directional type in Figure 1b says that if the initial goal has type $(\mathit{List}(1), \mathit{List}(1), 1)$, then either the result has type $(\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))$, i.e., the third component is bound to a list, or execution does not terminate successfully.

Consider the `append` predicate in Figure 1. An initial goal such as `append(1.2.nil,3.nil,C)` has type $(\mathit{List}(1), \mathit{List}(1), 1)$ since the first two arguments are lists and there is no restriction on the third argument. When `append` is used to split a list, an initial goal such as `append(A,B,1.2.3.nil)` has type $(1, 1, \mathit{List}(1))$ since the type admits any terms in the first two arguments and the third argument is a list.

We give a formal syntax and semantics for types using *set expressions* [MR85, HJ90b, AW92, BGW93] (Section 3). Set expressions can describe any regular set of tree-structured terms, including all of the standard recursive data types such as lists, binary trees, etc. Despite their great expressive power, many important properties of set expressions are decidable. This combination of expressiveness and computational tractability make set expressions a natural and attractive language for types.

Set expressions allow us to reason about the directionality of logic programs with non-ground goals as well as non-ground answers. For example, the type $\mathit{List}(1)$ says that the structure is that of a list, although the elements of the list can be anything (including variables). Previous work on modes for logic programs focussed on distinguishing whether or not predicate arguments could be bound to something other than a single variable (see Section 2). The types used in this paper distinguish not only whether an argument may be bound or unbound, but also describe the degree to which the argument is bound.

Section 4 uses directional types to define *well-typed* programs. Intuitively, a predicate p is well-typed with respect to a directional type $A \rightarrow B$ if for every $t_a \in A$, the goal $p\ t_a$ is resolved to a term $t_b \in B$. The main result of this paper is a procedure for verifying that a program is well-typed with respect to a given set of directional types. The essence of the procedure is to reduce the problem of checking well-typedness to a decision problem on *set constraints*, which are systems of inclusion constraints over set expressions. In Section 5 we present the reduction and prove its correctness.

When directional types are given using arbitrary set expressions, our procedure for well-typing is sound but not complete. That is, our procedure only accepts well-typed programs, but not all well-typed programs are accepted by our procedure. For the more restrictive class of *discriminative* directional types

[2] The complete definition is slightly more complex; see Section 4.

our procedure is both sound and complete: the procedure passes exactly the class of well-typed programs (see Section 6). While not as general as arbitrary directional types, discriminative directional types are still powerful enough to express all commonly used programming data types.

In addition to the algorithm for well-typing, we prove lower bounds on the complexity of the well-typing problem. For the general case, we show that the problem is EXPTIME-hard; for programs written with discriminative directional types we show that the problem remains PSPACE-hard.

The rest of this paper is organized as follows. Section 2 briefly surveys related work on modes and types. Section 3 introduces basic definitions and notation used throughout the paper. Sections 4 and 5 give a formal definition of well-typedness and the reduction of well-typedness to a decision problem on set constraints. Section 6 proves that the algorithm is a decision procedure for programs with discriminative directional types. The lower bound results are given in Section 7. Finally, Section 8 concludes with directions for future work.

2 Related Work

Our work touches on diverse research in type systems, mode systems, type checking and solving set constraints. We discuss the significant differences between our work this related research.

Type systems for logic programs such as [Mis84, MR85, HJ90a, FSVY91, HJ92, DZ92, YFS92] interpret types as sets of ground terms whereas we interpret types as sets of non-ground terms. Further, the above type systems do not address the issue of directionality, which is an integral part of the definition of directional types. Early works on mode systems have considered only simple modes such as ground/non-ground which do not specify the structure of terms. Subsequent work on moded type systems such as [Jac92, ZY92, JB92] do permit richer types but these types do not express the directionality of predicates.

The work of Heintze and Jaffar [HJ90a, HJ90b] deals with the more general problem of inference of types but over a more specific domain where types are interpreted as sets of ground terms. In addition, they do not deal with directionality. While their work also reduces the typing problem to a decision problem on set constraints, it is not clear if their technique can be adapted to our type checking problem.

The work of Rouzaud and Nguyen-Phong [RNP92] describes another type system based on set expressions. Though they view types as sets of non-ground terms, they require types to be *tuple-distributive sets*. The general algorithm described in this paper does not require types to be tuple-distributive and hence is able to describe more precise types. Their work does not give a description of the type checking algorithm nor does it provide a characterisation of the programs for which type checking can be done. Our work presents a simple transformation from the type checking problem to a decision problem on set constraints which not only enables us to give a simple description of type checking algorithm but also to characterise programs that can be type checked. Finally, we present bounds on the complexity of type checking, an issue that none of the above-mentioned works on type systems have dealt with.

The utility of the type checking algorithm is best illustrated by the work of Apt et al. [AE93, Apt93] which uses the directional type system [BLR92] to prove properties of well-typed programs. The type checking algorithm presented in this paper is a step towards automatic proofs of properties such as partial correctness [Apt93], compile-time optimizations [AE93] and termination [BLR92].

3 Definitions and Notation

We will be dealing with terms, substitutions on terms, set expressions, and substitutions on set expressions. To avoid confusion in subsequent sections, we define these concepts carefully here.

3.1 Terms

A *term* is defined by the grammar $t ::= X \mid c(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n)$, where X is a logic variable. Every constructor c has a fixed arity, which can be zero (a constant). A *ground* term contains no variables. We treat parenthesized terms without an associated constructor $(t_1, \ldots, t_n)$ as a distinguished constructor of arity n. There is a family of such constructors of arity zero $()$, one (t_1), two (t_1, t_2), etc. This convention allows us to write atoms $f(t_1, \ldots, t_n)$ as $f\ t$, where f is the predicate symbol and $t = (t_1, \ldots, t_n)$. Terms are denoted by $t, t_1, t_2, \ldots$. The set of all ground terms is H, the Herbrand Universe.

A *term substitution* is a function from variables to terms. A *ground* substitution is a function from variables to ground terms. Such substitutions extend in a straightforward manner to apply to terms. Substitutions on terms are denoted by lower-case Greek letters $\theta, \sigma, \ldots$. The *most general unifier* of two terms is denoted $mgu(t_1, t_2)$, if it exists. (A unifier is a substitution σ such that $\sigma(t_1) = \sigma(t_2)$.)

3.2 Logic Programs

A *clause* has the form $f_0\, t_0 \leftarrow \bigwedge_{1 \le i \le n} f_i\ t_i$, where f_j is a predicate symbol, and $t_j = (t_{j_1}, \ldots, t_{j_n})$ where n is the arity of predicate f_j. A *program* is a set of clauses. A *query* has the form $\bigwedge_{1 \le i \le n} f_i\ t_i$.

In keeping with the standard semantics of Prolog, we assume that subgoals are resolved in left-to-right order ("LD resolution"). This is a departure from pure logic programming but is consistent with logic programming languages used in practice.

3.3 Set Expressions

A *set expression* is defined by the following grammar:

$$T ::= \alpha \mid c(T_1, \ldots, T_n) \mid (T_1, \ldots, T_n) \mid T_1 \cup T_2 \mid T_1 \cap T_2 \mid \mathit{fix}\ \alpha.T_1 \mid 0 \mid 1$$

In this grammar, α is a set variable. Set expressions are denoted by capital Roman letters $A, B, \ldots$ or by $t, t_0, t_1, \ldots$ when the set expression is also a term. Set expressions can also include complement $\neg T_1$ [HJ90a] but these are not needed for the purposes of this paper.

A *set substitution* is a function from variables to sets of ground terms. Set substitutions are denoted by capital Greek letters $\Theta, \Sigma, \ldots$. A set expression together with a set substitution Θ denotes a set of ground terms, defined as follows:

$$\begin{aligned}
\Theta(c(T_1, \ldots, T_n)) &= \{c(t_1, \ldots, t_n) \mid t_i \in \Theta(T_i)\} \\
\Theta((T_1, \ldots, T_n)) &= \{(t_1, \ldots, t_n) \mid t_i \in \Theta(T_i)\} \\
\Theta(T_1 \cup T_2) &= \Theta(T_1) \cup \Theta(T_2) \\
\Theta(T_1 \cap T_2) &= \Theta(T_1) \cap \Theta(T_2) \\
\Theta(\mathit{fix}\ \alpha.T_1) &= \text{least } T \ \text{ s.t. } \ T = \Theta[\alpha \leftarrow T](T_1)
\end{aligned}$$

$$\Theta(0) = \emptyset$$
$$\Theta(1) = H$$

For example, consider the set expression *fix* $\alpha.(A.\alpha \cup \text{nil})$. Here we treat "." as an infix binary constructor. If we interpret "." as a list constructor and "nil" as the empty list then the meaning of this expression is $List(A)$, the set of all lists whose elements are drawn from A. Throughout the rest of this paper, $List(A)$ abbreviates the set expression *fix* $\alpha.(A.\alpha \cup \text{nil})$.

A variable in a set expression is *free* if it is not bound by a surrounding *fix*. A set expression with no free variables is *ground* and has the same meaning under all substitutions. For a ground set expression we drop the substitution and simply regard the expression as denoting a set of ground terms.

3.4 Systems of Set Constraints

A *system of set constraints* is a conjunction of constraints $\bigwedge_{1\leq i\leq n} A_i \subseteq B_i$ where the A_i and B_i are set expressions. A *solution* of the constraints is a substitution Θ such that for all i, $\Theta(A_i) \subseteq \Theta(B_i)$. The set of all solutions of a system S of constraints is denoted $Sol(S)$. For example, let $A = B$ stand for the system $A \subseteq B \wedge B \subseteq A$. Then the constraint $\alpha = b.\alpha \cup \text{nil}$ has a unique solution where $\alpha = \{\text{nil}, b.\text{nil}, b.b.\text{nil}, \ldots\}$, the set of all lists of b's.

For brevity, we refer to both term substitutions and set substitutions as "substitutions". The kind of substitution is always clear from the case of the Greek letter for the substitution (lower case for terms, upper case for set expressions). It is often useful to "lift" a ground term substitution to a set substitution. The lift of σ is written $\overline{\sigma}$ where $\overline{\sigma}(\alpha) = \{\sigma(\alpha)\}$.

4 Types and Well-Typing

We begin the development with a brief review of definitions of "type" and "well-typing" from [Apt93, BLR92, BLR93]. These definitions are independent of any particular representation of types.

Definition 4.1 A *type* is a set of terms closed under substitution.

Give a term t and type T, we write $t : T$, read "t has type T", if $t \in T$. The definition of a well-typed program relies on two subsidiary definitions of *type judgement* and *directional type*.

Definition 4.2 Let $s_1, \ldots, s_n, t$ be terms and let $S_1, \ldots, S_n, T$ be types. A *type judgement* has the form $\bigwedge_{1\leq i\leq n} s_i : S_i \Rightarrow t : T$. The judgement is true, written $\models \bigwedge_{1\leq i\leq n} s_i : S_i \Rightarrow t : T$, if for all substitutions σ, $\sigma(s_i) \in S_i$ for all i implies that $\sigma(t) \in T$.

A type judgement is just an implication that holds in all substitutions.

Definition 4.3 A *directional type* for a predicate f has the form $I \rightarrow O$ where I and O are types. The type I is the "input type" and type O is the "output type" of f.

Definition 4.3 is taken from [BLR92], where it is called *mode dependence*. In [Apt93], a more restrictive definition of directional type is given, where it is called a *type*.

Formally, a directional type is just a pair of types. Informally, however, it may be helpful to think of a directional type as an assertion that for any goal $f\ t$ where $t : I$, it follows that $\sigma(t) : O$ for any answer substitution σ. More succinctly, a directional type says that if the input is in I, then the output is in O.

A directional type for the `append` predicate in Figure 1 is

$$(List(1), List(1), 1) \rightarrow (List(1), List(1), List(1))$$

The dependence says that whenever the first two arguments of `append` are lists, then the third argument is resolved to a list.

Definition 4.4 (Well-Typed) Consider a program with directional types $I_j \rightarrow O_j$ for each predicate f_j.

- The clause $f_0\ t_0 \leftarrow \bigwedge_{1\leq i\leq n} f_i\ t_i$ is *well-typed* if the following two conditions hold:

$$\forall_{1\leq j\leq n} \models t_0 : I_0 \wedge \bigwedge_{1\leq i<j} t_i : O_i \Rightarrow t_j : I_j$$
$$\models t_0 : I_0 \wedge \bigwedge_{1\leq i\leq n} t_i : O_i \Rightarrow t_0 : O_0$$

- A program is well-typed if every clause is.
- A query $\bigwedge_{1\leq i\leq n} f_i\ t_i$ is well-typed if $\forall_{1\leq j\leq n} \models \bigwedge_{1\leq k<j} t_k : O_k \Rightarrow t_j : I_j$

LD-resolution is type consistent for well-typed programs [BLR92]. That is, given a well-typed program with directional types $I_j \rightarrow O_j$ for each predicate f_j and a well-typed query $\bigwedge_{1\leq i\leq n} f_i\ t_i$, for any answer substitution σ it is the case that $\bigwedge_{1\leq i\leq n} \sigma(t_i) : O_i$.

In general it is undecidable whether a program is well-typed, primarily because under Definition 4.1 types can be very rich sets. Thus, to get algorithms for deciding well-typing, it is necessary to restrict the set of permissible types. We explore the use of ground set expressions to denote types. A superficial problem with using ground set expressions for types is that the types contain only ground terms. This means, for example, that a variable X has no type. The following definition gives a more general interpretation of the type denoted by a ground set expression: a ground set expression stands for terms whose ground instances are in the set.

Definition 4.5 (Sat) The set of terms *satisfying* a ground set expression A, written $Sat(A)$, is

$$\{t | \forall \text{ground substitutions } \sigma.\ \sigma(t) \in A\}$$

Recall that the set expression 1 denotes the set of all ground terms. Therefore $Sat(1)$ is the set of all terms and for any term t we have $t : Sat(1)$. Define $List(X) = \mathit{fix}\ \alpha.(\text{nil} \cup \alpha.X)$. (Recall that "." is an infix binary constructor—see Section 3.) Since $List(1)$ is the set of all ground lists, any list term $t_1.\ldots.t_n.nil : Sat(List(1))$.

Lemma 4.6 For any ground set expression A, the set $Sat(A)$ is a type.

Proof: Let $t_1, t_2 \in Sat(A)$, and define $\sigma(\alpha) = t_2$ and $\sigma(\beta) = \beta$ for any variable $\beta \neq \alpha$. Now for any ground substitution θ, we have $\theta(\sigma(t_1)) \in A$, since $t_1 \in Sat(A)$ and $\theta \circ \sigma$ is a ground substitution. Therefore $\sigma(t_1) \in Sat(A)$. Since t_1 and t_2 were chosen arbitrarily, $Sat(A)$ is closed under substitution and is therefore a type. $\square$

For the remainder of this paper, all types are ground set expressions. For brevity in examples, we abuse our notation, writing $t : A$ instead of $t : Sat(A)$ and $A \to B$ instead of $Sat(A) \to Sat(B)$.

5 Type Checking

In this section we give a procedure for mapping the problem of checking that a program is well-typed to a decision problem on set constraints. The reduction takes as input a clause and a set of directional types, one for each predicate symbol in the clause, and produces conditions of the form $\mathcal{N}(S_1) \subseteq \mathcal{N}(S_2)$, where S_1 and S_2 are systems of set constraints and $\mathcal{N}(S_1)$ (defined below) is a certain subset of the solutions $Sol(S_1)$ of S_1.

The reduction we present is sound: whenever the conditions are true, the program is well-typed. If types are given by general set expressions, then the reduction is also conservative: if the conditions are false, then the program may or may not be well-typed. In Sections 6 we introduce *discriminative set expressions*, which are a subset of the set expressions but still expressive enough for many purposes. In the case where all types are discriminative, our algorithm is a decision procedure.

5.1 An Informal Example

Before presenting the formal development we give a high-level description of the algorithm. Informally, our algorithm reasons about well-typing as follows. Recall the program for the append predicate given in Figure 1. Let `append` have the directional type $(List(1), List(1), 1) \to (List(1), List(1), List(1))$. This type says that for a goal `append(A,B,C)`, if `A` and `B` are lists and the program succeeds, then `C` is instantiated to a list. The goal of the algorithm is to prove that `append` is well-typed with respect to this directional type.

To begin, consider only the first clause of `append`. If `(nil,X,X)` has the input type $(List(1), List(1), 1)$, then clearly `X` must be a list. To see this consider the components in order. For the first component, `nil` is a list, so `nil` satisfies $List(1)$. For the second component, `X` satisfies $List(1)$ only if `X` is a list of (possibly non-ground) terms. Finally, the third component of the input type imposes no constraints on `X`. The algorithm next checks that for every term that has the input type, the result has the output type. In this case it is easy to see that whenever `X` is a list, then all three components of the result are lists (i.e., `nil` and two occurrences of `X`), which satisfies the output type $(List(1), List(1), List(1))$.

For the second clause of `append` the chain of reasoning is longer but just as simple. The new wrinkle is the addition of subgoals. Our algorithm assumes that subgoals are well-typed and tries to prove that this implies that the clause is well-typed. If this can be done for every clause, then the program is well-typed.

Briefly, for the second clause of `append`, if the head of the clause (`W.X, Y, W.Z`) satisfies the input type $(List(1), List(1), 1)$, then `X` and `Y` must be lists. If `X` and `Y` are lists, then the subgoal `append(X, Y, Z)` satisfies the input type and (by the assumption that subgoals are well-typed) the result satisfies the output type $(List(1), List(1), List(1))$. If the result satisfies the output type, then `Z` is also a list. Finally, if `X`, `Y`, and `Z` are all lists, then (`W.X, Y, W.Z`) satisfies the output type $(List(1), List(1), List(1))$ so `append` is well-typed.

5.2 The General Case

In the general case, a clause has the form $f_0\ t_0 \leftarrow \bigwedge_{1 \leq i \leq n} f_i\ t_i$ with directional type $I_j \rightarrow O_j$ for each predicate f_j. By Definition 4.4 it is clear that we need consider only terms that have the input type $Sat(I_0)$. We can further restrict attention to those terms that unify with the head of the clause t_0, for otherwise this clause would not be selected. Thus, the first problem is to characterize the set of terms t such that $\sigma(t) : Sat(I_0)$ where $\sigma = mgu(t, t_0)$. Intuitively, this set is characterized by the solutions of $t_0 \subseteq I_0$, since the solutions include unifiers of t and t_0 that also satisfy I_0. However, the set $Sol(t_0 \subseteq I_0)$ may contain more solutions than necessary. For example a set constraint $\alpha.\beta \subseteq X$ has solutions where $\alpha = 0$ (the empty set), even though in any successful computation α must be bound to some term. We rule out solutions where a variable is assigned no terms:

Definition 5.1 If S is a set of constraints, the *non-zero solutions* of S, written $\mathcal{N}(S)$, are the solutions $\Theta \in Sol(S)$ such that $|\Theta(\alpha)| \geq 1$ for all α; i.e., each variable has at least one term.

Using this definition, the following lemma characterizes $Sat(A)$ in terms of constraints:

Lemma 5.2 Let t be a term, σ a substitution, and A a ground set expression.

$$\sigma(t) : Sat(A) \Leftrightarrow \forall \text{ ground } \theta.\ \overline{\theta \circ \sigma} \in \mathcal{N}(t \subseteq A)$$

Proof: Recall from Section 3.4 that $\overline{\sigma}(\alpha) = \{\sigma(\alpha)\}$ is the lift of a ground term substitution to a set substitution.

$$\begin{array}{lll} & \sigma(t) : Sat(A) & \\ \Leftrightarrow & \forall \text{ ground } \theta.\ \theta(\sigma(t)) \in A & \text{def of } Sat \\ \Leftrightarrow & \forall \text{ ground } \theta.\ \overline{\theta \circ \sigma}(t) \subseteq A & \text{def of lift} \\ \Leftrightarrow & \forall \text{ ground } \theta.\ \overline{\theta \circ \sigma}(t) \subseteq \overline{\theta \circ \sigma}(A) & \text{since } A \text{ is ground} \\ \Leftrightarrow & \forall \text{ ground } \theta.\ \overline{\theta \circ \sigma} \in \mathcal{N}(t \subseteq A) & \end{array}$$

The last line follows because $\theta \circ \sigma$ is a ground term substitution, and therefore the cardinality of every variable in the lifted substitution is one. □

Theorem 5.3 gives a sufficient condition for a program to be well-typed: one simply replaces the conditions in Definition 4.4 by the corresponding set constraint conditions.

Theorem 5.3

$$\mathcal{N}(\bigwedge_{1\leq i\leq n} t_i \subseteq A_i) \subseteq \mathcal{N}(\bigwedge_{1\leq j\leq m} s_j \subseteq B_j) \quad \Rightarrow \quad \models \bigwedge_{1\leq i\leq n} t_i : \mathit{Sat}(A_i) \Rightarrow \bigwedge_{1\leq j\leq m} s_j : \mathit{Sat}(B_j)$$

Proof: By Definition 4.2, the expression $\models \bigwedge_{1\leq i\leq n} t_i : \mathit{Sat}(A_i) \Rightarrow \bigwedge_{1\leq j\leq m} s_j : \mathit{Sat}(B_j))$ is equivalent to $\forall\sigma. \bigwedge_{1\leq i\leq n} \sigma(t_i) : \mathit{Sat}(A_i) \Rightarrow \bigwedge_{1\leq j\leq m} \sigma(s_j) : \mathit{Sat}(B_j)$. We reason as follows:

$$\begin{array}{lll} & \bigwedge_{1\leq i\leq n} \sigma(t_i) : \mathit{Sat}(A_i) & \\ \Leftrightarrow & \forall \text{ ground } \theta.\theta \circ \sigma \in \mathcal{N}(\bigwedge_{1\leq i\leq n} t_i \subseteq A_i) & \text{by applying Lemma 5.2 } n \text{ times} \\ \Rightarrow & \forall \text{ ground } \theta.\theta \circ \sigma \in \mathcal{N}(\bigwedge_{1\leq j\leq m} s_j \subseteq B_j) & \text{by assumption} \\ \Leftrightarrow & \bigwedge_{1\leq j\leq m} \sigma(s_j) : \mathit{Sat}(B_j) & \text{by applying Lemma 5.2 } m \text{ times} \end{array}$$

□

In the rest of this section we present two examples. First, we use Theorem 5.3 to prove that the `append` program in Figure 1 is well-typed. Second, we give an example showing that Theorem 5.3 is not a necessary condition for a program to be well-typed.

Returning to the `append` program in Figure 1, to check that `append` is well-typed we must check that the two clauses are well-typed. Since there are no subgoals in the first clause, the condition for well-typing (Definition 4.4) is

$$\models (\text{nil}, X, X) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), 1)) \Rightarrow (\text{nil}, X, X) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), \mathit{List}(1)))$$

Using Theorem 5.3, this condition holds if

$$\mathcal{N}((\text{nil}, X, X) \subseteq (\mathit{List}(1), \mathit{List}(1), 1)) \quad \subseteq \quad \mathcal{N}((\text{nil}, X, X) \subseteq (\mathit{List}(1), \mathit{List}(1), \mathit{List}(1)))$$

It is easy to check that the set of non-zero solutions of both systems is $\mathcal{N}(X \subseteq \mathit{List}(1))$, so the constraint holds. For the second clause, the conditions for well-typing are

$$\begin{array}{r} \models (W.X, Y, W.Z) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), 1)) \quad \Rightarrow \\ (X, Y, Z) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), 1)) \end{array}$$

$$\begin{array}{r} \models (W.X, Y, W.Z) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), 1)) \wedge (X, Y, Z) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))) \quad \Rightarrow \\ (W.X, Y, W.Z) : \mathit{Sat}((\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))) \end{array}$$

Using Theorem 5.3 again, we get

$$\begin{array}{r} \mathcal{N}((W.X, Y, W.Z) \subseteq (\mathit{List}(1), \mathit{List}(1), 1)) \quad \subseteq \\ \mathcal{N}((X, Y, Z) \subseteq (\mathit{List}(1), \mathit{List}(1), 1)) \end{array}$$

$$\begin{array}{r} \mathcal{N}((W.X, Y, W.Z) \subseteq (\mathit{List}(1), \mathit{List}(1), 1 \ \wedge \ (X, Y, Z) \subseteq (\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))) \quad \subseteq \\ \mathcal{N}((W.X, Y, W.Z) \subseteq (\mathit{List}(1), \mathit{List}(1), \mathit{List}(1))) \end{array}$$

For the first condition, it is easy to see that the set of non-zero solutions of both systems is $\{X \subseteq List(1) \wedge Y \subseteq List(1)\}$. For the second condition, the set of non-zero solutions of both systems is $\{X \subseteq List(1) \wedge Y \subseteq List(1) \wedge Z \subseteq List(1)\}$. Since both conditions hold, the second clause of `append` is well-typed. Since both clauses are well-typed, `append` is well-typed.

While Theorem 5.3 is sufficient, it is not a necessary condition, so the reduction is only conservative: if the set constraint conditions hold, then the program is well-typed, otherwise it may or may not be well-typed. Section 6 gives a necessary and sufficient condition for a more restrictive class of set constraints, the *discriminative* constraints. The following example shows why Theorem 5.3 is not enough to prove that a program is well-typed.

Example 5.4 Consider the program

$$\mathtt{p(X,X)} \leftarrow \mathtt{p(X,X)}$$

and let `p` have directional type

$$(a \cup b, a \cup b) \rightarrow (a, a) \cup (b, b)$$

By Definition 4.4, this program is well-typed if

$$\models p(X, X) : (a \cup b, a \cup b) \Rightarrow p(X, X) : (a, a) \cup (b, b)$$

The program is well-typed, because the only two terms in the input type that match the head of the clause are `(a,a)` and `(b,b)` and both of these terms are in the output type. Converting the condition above to a set constraint condition using Theorem 5.3, we get:

$$\mathcal{N}((X, X) \subseteq (a \cup b, a \cup b)) \subseteq \mathcal{N}((X, X) \subseteq (a, a) \cup (b, b))$$

This is false, since $X = a \cup b$ is a solution of the first system but not of the second.

6 Discriminative Constraints

This section introduce discriminative types and proves that our algorithm is a decision procedure for well-typing if all types are discriminative.

Definition 6.1 The *discriminative set expressions* are the smallest set D satisfying:

- $\{0, 1\} \subseteq D$
- $\alpha \in D$ for every variable α
- $f(x_1, \ldots, x_n) \in D$ if $\forall_{1 \leq i \leq n}\ x_i \in D$
- $\mathit{fix}\ \alpha.x \in D$ if $x \in D$
- $\bigcup_{1 \leq i \leq n} f_i(x_{i_1}, \ldots, x_{i_n}) \in D$ if $\forall_{1 \leq i \leq n}\ f_i(x_{i_1}, \ldots, x_{i_n}) \in D$ and $f_i \neq f_j$ for $i \neq j$.

The important restrictions of discriminative set expressions are that there are no intersection operations and all unions are formed from expressions with distinct outermost constructors. Commonly used data types can be described as discriminative set expressions. For example, the type $List(A) = \mathit{fix}\ \alpha.(\mathrm{nil} \cup \alpha.A)$ is discriminative whenever A is discriminative.

Definition 6.2 A *system of discriminative set constraints* is a conjunction of constraints $\bigwedge_{1 \le i \le n} A_i \subseteq B_i$ where the A_i and B_i are discriminative set expressions.

Theorem 6.3 Let $var(t)$ be the set of variables in term t. Let $t_1, \ldots, t_n$ be terms, $A_1, \ldots, A_n$ be discriminative ground set expressions, and let $var(t_1) \cup \ldots \cup var(t_n) = \{X_1, \ldots, X_m\}$. Then

$$\mathcal{N}(\bigwedge_{1 \le i \le n} t_i \subseteq A_i) = \mathcal{N}(\bigwedge_{1 \le j \le m} X_j \subseteq B_j)$$

for some ground set expressions $B_1, \ldots, B_m$. Furthermore, $B_1, \ldots, B_m$ are computable.

The proof of Theorem 6.3 follows from analysis of an algorithm to solve a more general class of set constraints in [MR85]. The advantage of using discriminative set expressions for types is that the solutions of the set constraints have more structure. We require the following definition:

Definition 6.4 Let Z be a set of ground substitutions. $(\bigsqcup Z)(\alpha) = \bigcup_{\sigma \in Z} \sigma(\alpha)$

Given two ground substitutions σ and σ', define $\sigma \le \sigma'$ if $\sigma \sqcup \sigma' = \sigma'$.

Lemma 6.5 Let Z be a set of ground substitutions and let $S = \bigwedge_{1 \le i \le m} \alpha_i \subseteq A_i$ where A_i is ground. Then

$$Z \subseteq Sol(S) \Leftrightarrow \bigsqcup Z \in Sol(S)$$

Proof: Let $Z \subseteq Sol(S)$. Then $(\bigsqcup Z)(\alpha_i) = \bigcup_{\sigma \in Z} \sigma(\alpha_i) \subseteq A_i$ since $\sigma(\alpha_i) \subseteq A_i$ for all $\sigma \in Z$. For the other direction, assume $\bigsqcup Z \in Sol(S)$ and let $\sigma \in Z$. Then $\sigma(\alpha) \subseteq (\bigsqcup Z)(\alpha_i) \subseteq A_i$. $\square$

Lemma 6.5 says that the solutions of the set constraints in Theorem 6.6 are closed under upper bounds if types are discriminative. This is enough to make the reduction of well-typing to set constraints a necessary and sufficient condition.

Theorem 6.6 Let $A_1, \ldots, A_n, B_1, \ldots, B_m$ be discriminative ground set expressions. Then

$$\mathcal{N}(\bigwedge_{1 \le i \le n} t_i \subseteq A_i) \subseteq \mathcal{N}(\bigwedge_{1 \le j \le m} s_j \subseteq B_j) \quad \Leftrightarrow \quad \models \bigwedge_{1 \le i \le n} t_i : Sat(A_i) \Rightarrow \bigwedge_{1 \le j \le m} s_j : Sat(B_j)$$

Proof: The forward direction follows from Theorem 5.3. For the backward direction,

$$\begin{array}{lll}
 & \sigma \in \mathcal{N}(\bigwedge_{1 \le i \le n} t_i \subseteq A_i) & \\
\Leftrightarrow & \bigsqcup\{\overline{\sigma_0} | \overline{\sigma_0} \le \sigma\} \in \mathcal{N}(\bigwedge_{1 \le i \le n} t_i \subseteq A_i) & \text{since } \sigma = \bigsqcup\{\overline{\sigma_0} | \overline{\sigma_0} \le \sigma\} \\
\Leftrightarrow & \forall_{\overline{\sigma_0} \le \sigma}\ \overline{\sigma_0} \in \mathcal{N}(\bigwedge_{1 \le i \le n} t_i \subseteq A_i) & \text{by Lemma 6.5} \\
\Leftrightarrow & \forall_{\overline{\sigma_0} \le \sigma}\ \bigwedge_{1 \le i \le n} \overline{\sigma_0}(t_i) : Sat(A_i) & \text{by Lemma 5.2 using } \overline{\sigma_0} \text{ is ground} \\
\Rightarrow & \forall_{\overline{\sigma_0} \le \sigma}\ \bigwedge_{1 \le j \le m} \overline{\sigma_0}(s_j) : Sat(B_j) & \text{by assumption} \\
\Leftrightarrow & \forall_{\overline{\sigma_0} \le \sigma}\ \overline{\sigma_0} \in \mathcal{N}(\bigwedge_{1 \le j \le n} s_j \subseteq B_j) & \text{by Lemma 5.2 using } \overline{\sigma_0} \text{ is ground} \\
\Leftrightarrow & \bigsqcup\{\overline{\sigma_0} | \overline{\sigma_0} \le \sigma\} \in \mathcal{N}(\bigwedge_{1 \le j \le n} s_j \subseteq B_j) & \text{by Lemma 6.5} \\
\Leftrightarrow & \sigma \in \mathcal{N}(\bigwedge_{1 \le j \le n} s_j \subseteq B_j) & \text{since } \sigma = \bigsqcup\{\overline{\sigma_0} | \overline{\sigma_0} \le \sigma\}
\end{array}$$

□

7 Complexity

This section presents three results. First, we show that predicates of the form $\mathcal{N}(A) \subseteq \mathcal{N}(B)$ are decidable for arbitrary systems of set constraints A and B. This shows that Theorem 5.3 gives a semi-decision procedure for well-typing when types are given by ground set expressions, and (by Theorem 6.6) a decision procedure for well-typing when types are given by discriminative ground set expressions. Since the satisfiability of set constraints is complete for NEXPTIME, this gives an NEXPTIME upper bound for the discriminative case. Second, we show that well-typing where types are arbitrary ground set expressions is hard for EXPTIME. Third, we show that well-typing where types are discriminative ground set expressions is still hard for PSPACE. The exact complexity of the discriminative case remains open; for the non-discriminative case, no upper bound is known.

For set expressions s_1 and s_2, a *negative constraint* has the form $s_1 \not\subseteq s_2$ and $Sol(s_1 \not\subseteq s_2) = \{\sigma | \sigma(s_1) \not\subseteq \sigma(s_2)\}$. For a systems of set constraints A and B, let $Sol(A) \cap Sol(B)$, $Sol(A) \cup Sol(B)$, and $\sim Sol(A)$ denote the intersection, union, and complement of the solution sets respectively. The following theorem is proven in [AKW93, GTT93].

Theorem 7.1 Let A be any boolean combination (i.e., intersection, union, or complement) of systems of set constraints with positive and negative constraints. It is decidable whether A denotes the empty set of solutions.

We use Theorem 7.1 to show that the predicate $\mathcal{N}(A) \subseteq \mathcal{N}(B)$ is decidable. First, $\mathcal{N}(A) = Sol(A) \cap Sol(\bigwedge_{1 \le i \le n} X_i \not\subseteq 0)$ where $var(A) = \{X_1, \ldots, X_n\}$. Thus, $\mathcal{N}()$ can be replaced by $Sol()$ with some additional negative constraints. To finish, note that $Sol(X) \subseteq Sol(Y)$ if and only if $Sol(X) \cap \sim Sol(Y) = \emptyset$.

Theorem 7.2 If types are given by ground set expressions, well-typing is hard for EXPTIME.

Proof: [sketch] Consider the program $P(X) \leftarrow P(X)$ with directional type $I \rightarrow O$. The program is well-typed iff $O \subseteq I$. For every tree automaton T, there is a ground set expression S (with size polynomial in the size of the automaton) such that the language accepted by T is the set denoted by S. Thus, testing inclusion of ground set expressions is at least as hard as testing inclusion of languages accepted by tree automata, which is EXPTIME-complete [Sei90]. □

Theorem 7.3 If types are given by discriminative ground set expressions, well-typing is hard for PSPACE.

Proof: [sketch] It is known that testing whether the intersection of n deterministic finite automata (DFA) is non-empty is PSPACE-complete [Koz77]. For every DFA T, there is a discriminative set expression S (with size polynomial in the size of the automaton) such that the language accepted by T is the set denoted by S. Consider the program $P(X, \ldots, X) \leftarrow P(X, \ldots, X)$ where X is repeated n times. Let the directional type be $(T_1, \ldots, T_n) \rightarrow 0$, where T_i is an encoding of a DFA. The program is well-typed if and only if $T_1 \cap \ldots \cap T_n = 0$. □

8 Conclusions and Future Work

Set expressions provide a very expressive framework for defining directional types. In this paper, we have shown that type checking of directionally typed logic programs can be reduced naturally to a decision problem on set constraints. Type checking is performed by a separate analysis of each clause in the program. The examples in this paper have been checked by running our algorithm by hand. We hope to implement the algorithm to find out how useful it is on large logic programs.

An obvious area for future work is to perform type inference instead of type checking; that is, to automatically infer the types used without the need for the programmer to supply directional types for predicates. This seems like a difficult problem, since a predicate can have many directional types described by set expressions, only a few of which are probably interesting. It is not clear how to automatically identify the "right" directional types for a predicate.

Another direction for future work is to explore applying these techniques to other problems in logic programming that depend on type information. For example, the techniques for analyzing control in sequential [Nai86], concurrent [Sha89] and parallel [Gre87] logic programming languages rely on knowledge about predicate types.

Acknowledgements

We would like to thank Saumya Debray, Nevin Heintze, Uday Reddy, Moshe Vardi, and Ed Wimmers for discussions and comments on some of the ideas presented in this paper.

References

[AE93] K. R. Apt and Sandro Etalle. On the Unification-free Prolog Programs. In *Proceedings of the 1993 Conference on Mathematical Foundations of Computer Science*, June 1993.

[AKW93] A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. Technical Report 93-1362, Cornell University, June 1993.

[Apt93] K. R. Apt. Declarative Programming in Prolog. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium*. MIT Press, November 1993.

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[BGW93] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Symposium on Logic in Computer Science*, pages 75–83, June 1993.

[BLR92] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, pages 321–335, November 1992.

[BLR93] F. Bronsard, T. K. Lakshman, and U. S. Reddy. Directionally Typed Prolog: Unifying notions of Types and Directionality. Technical Report submitted to ICLP '94 and available via anonymous ftp from a.cs.uiuc.edu: in directory pub/reddy/tkl, November 1993.

[Deb89] S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

[DZ92] P. W. Dart and J. Zobel. *A Regular Type Language for Logic Programs*, chapter in *Types in Logic Programming*, Frank Pfenning (ed.), pages 157–187. MIT Press, Cambridge, MA, 1992.

[FSVY91] T. Früwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Symposium on Logic in Computer Science*, pages 300–309, July 1991.

[Gre87] S. Gregory. *Parallel Logic Programming in PARLOG.* Addison-Wesley, 1987.

[GTT93] R. Gilleron, S. Tison, and M. Tommasi. Solving Systems of Set Constraints with Negated Subset Relationships. In *Foundations of Computer Science*, pages 372–380, November 1993.

[HJ90a] N. Heintze and J. Jaffar. A Finite Presentation Theorem for approximating logic programs. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.

[HJ90b] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[HJ92] N. Heintze and J. Jaffar. *Semantic Types for Logic Programs*, chapter in *Types in Logic Programming*, Frank Pfenning (ed.), pages 141–155. MIT Press, Cambridge, MA, 1992.

[Jac92] D. Jacobs. *A Pragmatic View of Types for Logic Programs*, chapter in *Types in Logic Programming*, Frank Pfenning (ed.), pages 217–228. MIT Press, Cambridge, MA, 1992.

[JB92] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2):205–258, July 1992.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *IEEE Symposium on the Foundations of Computer Science*, pages 254–266. IEEE Computer Society, 1977.

[Mis84] P. Mishra. Towards a theory of types in PROLOG. In *Proceedings of the First IEEE Symposium in Logic Programming*, pages 289–298, 1984.

[MR85] P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, January 1985.

[Nai86] Lee Naish. *Negation and control in Prolog.* Lecture Notes in Computer Science 238. Springer-Verlag, 1986.

[RNP92] Yann Rouzaud and Lan Nguyen-Phoung. Integrating Modes and Subtypes into a Prolog Type-Checker. In *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, pages 85–97, November 1992.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, May 1989.

[YFS92] E. Yardeni, T. Fruhwirth, and E. Shapiro. *Polymorphically Typed Logic Programs*, chapter in *Types in Logic Programming*, Frank Pfenning (ed.), pages 63–90. MIT Press, Cambridge, MA, 1992.

[ZY92] J. L. Zachary and K. Yelick. *Moded Type Systems to Support Abstraction*, chapter in *Types in Logic Programming*, Frank Pfenning (ed.), pages 217–228. MIT Press, Cambridge, MA, 1992.