Directional Type Inference for Logic Programs

Witold Charatonik* and Andreas Podelski

Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken
{witold;podelski}@mpi-sb.mpg.de

Abstract. We follow the set-based approach to directional types proposed by Aiken and Lakshman [1]. Their type *checking* algorithm works via set constraint solving and is sound and complete for given discriminative types. We characterize directional types in model-theoretic terms. We present an algorithm for *inferring* directional types. The directional type that we derive from a logic program $\mathcal{P}$ is uniformly at least as precise as any discriminative directional type of $\mathcal{P}$, i.e., any directional type out of the class for which the type *checking* algorithm of Aiken and Lakshman is sound and complete. We improve their algorithm as well as their lower bound and thereby settle the complexity (DEXPTIME-complete) of the corresponding problem.

1 Introduction

Directional types form a type system for logic programs which is based on the view of a predicate as a *directional procedure* which, when applied to a tuple of input terms, generates a tuple of output terms. There is a rich literature on types and directional types for which we can give only some entry points. Directional types occur as predicate profiles in [28], as mode dependencies in [8], and simply as types in [4, 2, 3]. Our use of the terminology "directional type" stems from [1].

A directional type for a program $\mathcal{P}$ assigns input types I_p and output types O_p to each predicate p of $\mathcal{P}$. A program can have many directional types. For example, consider the predicate *append* defined by

$$\begin{array}{l} append([], L, L). \\ append([X|Xs], Y, [X|Z]) \leftarrow append(Xs, Y, Z). \end{array}$$

We can give this predicate the directional type $(list, list, \top) \rightarrow (list, list, list)$, where *list* denotes the set of all lists and $\top$ is the set of all terms, but also $(\top, \top, list) \rightarrow (list, list, list)$, as well as the least precise type $(\top, \top, \top) \rightarrow (\top, \top, \top)$. A predicate defined by a single fact $p(X)$ has a directional type $\tau \rightarrow \tau$ for all types τ.

In [1], Aiken and Lakshman present an algorithm for automatic type checking of logic programs wrt. given directional types. The algorithm runs in

* On leave from University of Wrocław, Poland. Partially supported by Polish KBN grant 8T11C02913.

NEXPTIME; they show that the problem is DEXPTIME-hard in general and PSPACE-hard for discriminative types. The algorithm works via set constraint solving; its correctness relies on a connection between the well-typedness conditions and the set constraints to which they are translated. The connection is such that the type check is sound and complete for *discriminative* types (it is still sound for general types).

Our results. In this paper, we answer two questions left open in [1]. First, we give an algorithm for *inferring* directional types. Second, we establish the DEXPTIME-completeness of the problem of directional type *checking* wrt. discriminative types.

To fix just one (the "right" one) directional type for a given logic program, we assume that the program comes with a query which, wlog., consists of one atom $main(t)$. Clearly, the choice of the type makes sense only if the input type I_{main} for the query predicate $main$ contains at least the expected set of input terms for $main$. Ideally, among all those directional types $\mathcal{T}$ that satisfy this condition, we would like to infer the uniformly (i.e., for the input types and output types of *all* predicates) most precise one.

The uniformly most precise directional type $\mathcal{T}_{min}(\mathcal{P})$ of a program $\mathcal{P}$ together with a specification of the query input terms does exist, as we will show. It is, however, not effectively computable in general. This is naturally the place where *abstraction* comes in. We can compute a directional type $\mathcal{T}_{sb}(\mathcal{P})$, a regular approximation of $\mathcal{T}_{min}(\mathcal{P})$ which is defined through the *set-based* abstraction à la Heintze and Jaffar [24]. There is no objective criterion to evaluate the quality of the approximation of a non-regular set by a regular one in the sense that the most precise approximation does not exist; this fact applies also to our type inference procedure. We can show, however, that $\mathcal{T}_{sb}(\mathcal{P})$ is uniformly more precise than any discriminative directional type of $\mathcal{P}$, i.e., any directional type out of the class for which the type *check* of Aiken and Lakshman is sound and complete.

The above comparison is interesting for intrinsic reasons and it indicates that our type inference procedure produces "good" directional types. We exploit it furthermore in order to derive a type *checking* algorithm whose complexity improves upon the one of the original algorithm in [1]. A simple refinement of the arguments given in [1] suffices to make the lower bound more precise. We thus settle the complexity (DEXPTIME-complete) of the problem of directional type checking of logic programs wrt. discriminative types.

Technically, our results are based on several basic properties of three kinds of abstraction (and their interrelation): the *set-based* abstraction (obtained by Cartesian approximation and related to our inference procedure), the *set-valued* abstraction (obtained by replacing membership constraints with set inclusions and related to the type-checking procedure for arbitrary types), and *path closure* abstraction (related to the type-checking procedure for discriminative types). These properties, that we collect in Section 2, are of general interest; in particular, the abstraction by path closure keeps reappearing (see, e.g., [27, 26, 31, 19, 20]). Furthermore, we establish that the directional types of a program $\mathcal{P}$ are exactly the models of an associated logic program $\mathcal{P}_{InOut}$. In fact, $\mathcal{P}_{InOut}$ is a kind of "magic set transformation" (see, e.g., [20]) of $\mathcal{P}$. We obtain our results (and the soundness and completeness results in [1]) by combining the model-theoretic characterization of directional types with the properties of abstractions established in Section 2. In fact, by having factored out general properties of abstractions from the aspects proper to directional types, we have maybe given a new view of the results in [1].

Other related work. Rouzaud and Nguyen-Phong [28] describe a type system where types are sets of non-ground terms and express directionality, but these sets must be tuple-distributive. Our types need not be tuple-distributive. In [14], Codish and Demoen infer type dependencies for logic programs. Their techniques (abstract compilation) are quite different from ours and the derived dependencies express all possible input-output relationships. Probably the work of Heintze and Jaffar is the one that is most closely related to ours. This is not only due to the fact that we use set-based analysis [24] to approximate the type program. Some of their papers [23, 25] contain examples where they compute for each predicate p a pair of sets $Call_p$ and Ret_p, which can be viewed as (ground) directional type $Call_p \rightarrow Ret_p$. We are not aware, however, of a general, formal treatment of directional types inference in their work. Also the work of Gallagher and de Waal [20, 21] is closely related to ours. They use a kind of "magic set transformation" to compute query and answer predicates for each predicate in the given program, which is essentially the same as our type program. Then they approximate the success set of the new predicates with ground, regular and tuple-distributive sets, which they do not call types. Boye in [5] (see also [6, 7]) presents a procedure that infers directional types for logic programs. The procedure is not fully automatic (sometimes it requires an interaction with the user), it requires the set of possible types to be finite, and no complexity analysis is given. In our approach, any regular set of terms is an admissible type; our procedure is fully automatic in the presence of a query for the program (or lower bounds for input types) and it runs in single-exponential time. We refer to [1] for comparison with still other type systems. Most of those interpret types as sets of ground terms, while we interpret types as sets of non-ground terms. Most type systems do not express the directionality of predicates.

2 Abstractions

In this section we discuss three kinds of abstraction of a given program. The motivation for this discussion is the following. In the next section we will define a type program for a given program $\mathcal{P}$. The models of this program are directional types of $\mathcal{P}$. The least model of the *set-based* abstraction of the type program correspond to the type that we infer. The models of the *set-valued* abstraction correspond to the directional types that pass the type-check from [1]. The models of the *path-closure* abstraction correspond to discriminative directional types of $\mathcal{P}$. Later we will use the relations between these abstractions to compare different directional types of a given program.

Preliminaries. We follow the notation and terminology of [1] unless specified otherwise. For example, we use the symbols $p, q, p_0, \ldots$ for predicates (instead of $f, g, f_0, \ldots$ as in [1]) and write $p(t)$ for predicate atoms (instead of $f\ t$). Wlog., all predicates are unary. Terms t are of the form x or $f(t_1, \ldots, t_n)$. We may write $t[x_1, \ldots, x_m]$ for t if $x_1, \ldots, x_m$ are the variable occurrences in t (we distinguish between multiple occurrences of the same variable), and $t[t_1, \ldots, t_m]$ for the term obtained from t by substituting t_j for the occurrence x_j. A logic program $\mathcal{P}$ is a set of Horn clauses, i.e., implications of the form $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$. A program comes with a query ("the main loop") which, wlog., is specified by one atom $main(t)$. The interpretation of programs, which is defined as usual, may be viewed as a mapping from predicate symbols to sets of trees. Programs may be viewed as formulas whose free variables are set-valued (and are referred to via predicate symbols), hence as a large class of *set constraints*.

We will not use (positive) set expressions and set constraints as in [1] but, instead, logic programs, for specifying sets of trees as well as for abstraction. Positive set expressions, which denote sets of trees, can be readily translated into equivalent *alternating* tree automata, which again form the special case of logic programs whose clauses are all of the form

$$p(f(x_1, \ldots, x_n)) \leftarrow p_{11}(x_1), \ldots, p_{1m_1}(x_1), \ldots, p_{n1}(x_n), \ldots, p_{nm_n}(x_n)$$

where $x_1, \ldots, x_n$ are pairwise different. A *non-deterministic* tree automaton is the special case where $m_1 = \ldots = m_n = 1$, i.e., a logic program whose clauses are all of the form $p(f(x_1, \ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n)$. A set of trees is *regular* if it can be denoted by a predicate p in the least model of a non-deterministic tree automaton.

A *uniform program* [19] consists of Horn clauses in one of the following two forms. (In a *linear* term, each variable occurs at most once.)

- $p(t) \leftarrow p_1(x_1), \ldots, p_k(x_m)$ where the term t is linear.
- $q(x) \leftarrow p_1(t_1), \ldots, p_m(t_m)$ where $t_1, \ldots, t_m$ are any terms over $\Sigma(\mathsf{Var})$.

A uniform program can be transformed (in single-exponential time) into an equivalent non-deterministic tree automaton [19, 18, 10].

Set-based abstraction. We use set-based analysis in the sense of [24] but in the formulation using logic programs as in [19, 18, 10] (instead of set constraints as in [24] and [1]).

Definition 1 ($\mathcal{P}^{\#}$, the set-*based* abstraction of $\mathcal{P}$). *The uniform program $\mathcal{P}^{\#}$ is obtained from a program $\mathcal{P}$ by translating every clause $p(t) \leftarrow$ body, whose head term t contains the n variables $x_1, \ldots, x_n$, into the $(n+1)$ clauses*

$$p(\tilde{t}) \leftarrow p_1(x_1^1), \ldots, p_1(x_1^{m_1}), \ldots, p_n(x_n^1), \ldots, p_n(x_n^{m_n})$$
$$p_i(x_i) \leftarrow \text{body} \quad \textit{(for } i = 1, \ldots, n\textit{)}$$

where $\tilde{t}$ is obtained from t by replacing the m_i different occurrences of variables x_i by different renamings $x_i^1, \ldots, x_i^{m_i}$, and $p_1, \ldots, p_n$ are new predicate names.

The least model of the program $\mathcal{P}^\#$ expresses the *set-based* abstraction of $\mathcal{P}$, which is defined in [24] as the least fixpoint of the operator $\tau_\mathcal{P}$. The operator $\tau_\mathcal{P}$ is defined via set-based substitutions in [24]; it can also be defined by (for a subset M of the Herbrand base)

$$\begin{aligned}\tau_\mathcal{P}(M) = \{p_0(t_0[x_1\theta_1, \dots, x_m\theta_m]) \mid\ & p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n) \in \mathcal{P},\\ & \theta_1, \dots, \theta_m \ \text{ ground substitutions},\\ & p_1(t_1\theta_1), \dots, p_n(t_n\theta_1) \in M\\ & \qquad\qquad \vdots\\ & p_1(t_1\theta_m), \dots, p_n(t_n\theta_m) \in M\}\end{aligned}$$

where $x_1, \dots, x_m$ are the variable occurrences in t_0 (we distinguish between multiple occurrences of the same variable). As noted in [19], the logical consequence operator associated with $\mathcal{P}^\#$ is equal to the set-based consequence operator; i.e., $T_{\mathcal{P}^\#} = \tau_\mathcal{P}$. We recall that the logical consequence operator associated with the program $\mathcal{P}$ is defined by

$$\begin{aligned}T_\mathcal{P}(M) = \{p_0(t_0[x_1\theta, \dots, x_m\theta] \mid\ & p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n) \in \mathcal{P},\\ & \theta \ \text{ ground substitution},\\ & p_1(t_1\theta), \dots, p_n(t_n\theta) \in M\}.\end{aligned}$$

The set-based abstraction can be formalized in the abstract interpretation framework [16] by the application of the *Cartesian approximation* $\mathcal{C}$ to the semantics-defining fixpoint operator $T_\mathcal{P}$; i.e., $T_{\mathcal{P}^\#} = \mathcal{C}(T_\mathcal{P})$ (see [17]; roughly, $\mathcal{C}$ maps a set of tuples to the smallest Cartesian product containing it). Thus, we have

$$T_{\mathcal{P}^\#} = \tau_\mathcal{P} = \mathcal{C}(T_\mathcal{P}).$$

We note the following fact, keeping symmetry with Remarks 2 and 3 on the two other abstractions of $T_\mathcal{P}$ that we will introduce. Given two set-valued functions F and F', we write $F \leq F'$ if F is smaller than F' wrt. to pointwise subset inclusion, i.e., $F(x) \subseteq F'(x)$ for all x.

Remark 1 (Set-based approximation). The direct-consequence operator associated with $\mathcal{P}^\#$ approximates the one associated with $\mathcal{P}$; i.e.,

$$T_\mathcal{P} \leq T_{\mathcal{P}^\#}.$$

Proof. Obvious by definition. □

The following statement will be used for the soundness of our type inference algorithm (Theorem 3). Its converse does, of course, not hold in general (the least models of $\mathcal{P}$ may be strictly smaller than the least models of $\mathcal{P}^\#$).

Proposition 1. *Each model $\mathcal{M}$ of the set-based abstraction $\mathcal{P}^\#$ of a program $\mathcal{P}$ is also a model of $\mathcal{P}$.*

Proof. A ground instance of a clause of $\mathcal{P}$ is also a ground instance of the corresponding clause of $\mathcal{P}^{\#}$. $\square$

Set-valued abstraction. The second abstraction that we consider is also defined via a program transformation: an atom $p(t)$ is simply replaced by an inclusion $t \subseteq p$. The transformed program is interpreted over the domain of *sets* of trees; i.e., the valuations are mappings $\theta : \mathsf{Var} \rightarrow 2^{T_\Sigma}$. These mappings are extended canonically from variables x to terms t; i.e., $t\theta$ is a set of trees. We repeat that an interpretation $\mathcal{M}$, i.e., a subset of the Herbrand base, maps predicates p to sets of trees $p^{\mathcal{M}} = \{t \in T_\Sigma \mid p(t) \in \mathcal{M}\}$. The inclusion $t \subseteq p$ holds in $\mathcal{M}$ under the valuation θ if $t\theta$ is a subset of $p^{\mathcal{M}}$.

Definition 2 **($\mathcal{P}^{\subseteq}$, the set-*valued* abstraction of $\mathcal{P}$).** *Given a program $\mathcal{P}$, its set-valued program abstraction is a program $\mathcal{P}^{\subseteq}$ that is interpreted over sets of trees (instead of trees). It is obtained by replacing membership with subset inclusion; i.e., it contains, for each clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ in $\mathcal{P}$, the implication*

$$t_0 \subseteq p_0 \;\leftarrow\; t_1 \subseteq p_1, \ldots, t_n \subseteq p_n. \tag{1}$$

The models of the program $\mathcal{P}^{\subseteq}$ are, as one expects, interpretations $\mathcal{M}$ (subsets of the Herbrand base) such that all implications all valid in $\mathcal{M}$. An implications of the form (1) is valid in $\mathcal{M}$ if for all valuations $\theta : \mathsf{Var} \rightarrow 2^{T_\Sigma}$, if $t_i\theta$ is a subset of $p_i^{\mathcal{M}}$ for $i = 1, \ldots, n$ then also for $i = 0$.

The models of $\mathcal{P}^{\subseteq}$ are the fixpoints of $T_{\mathcal{P}^{\subseteq}}$, the direct consequence operator associated with $\mathcal{P}^{\subseteq}$, which is defined in a way analogous to $T_{\mathcal{P}}$ (using set-valued substitutions instead of tree-valued substitutions). Hence, we will be able to use the following remark when we compare models of $\mathcal{P}^{\#}$ with models of $\mathcal{P}^{\subseteq}$ (Proposition 4).

Remark 2 (Set-valued approximation). The direct-consequence operator associated with $\mathcal{P}^{\subseteq}$ approximates the one associated with $\mathcal{P}$; i.e.,

$$T_{\mathcal{P}} \leq T_{\mathcal{P}^{\subseteq}}.$$

Proof. If, for some subset M of the Herbrand base, $T_{\mathcal{P}}(M)$ contains the ground atom $p_0(t_0)$ because M contains the ground atoms $p_1(t_1), \ldots, p_n(t_n)$ and $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ is a ground instance of some clause of $\mathcal{P}$, then the singleton $\{t_i\}$ is a subset of the denotation of p_i under M (for $i = 1, \ldots, n$) and, hence, the singleton $\{t_0\}$ is a subset of the denotation of p_0 under $T_{\mathcal{P}^{\subseteq}}(M)$. $\square$

The next statement underlies the *soundness* of the type checking procedure of [1], which essentially checks if a given interpretation is a model of the program $\mathcal{P}^{\subseteq}_{InOut}$ defined in the next section (cf. Theorem 9 in [1]). It says that being a model wrt. $\mathcal{P}^{\subseteq}$ is a sufficient condition for being a model of the program $\mathcal{P}$. (The model property of a regular set wrt. $\mathcal{P}^{\subseteq}$ is nothing else than an entailment relation between set constraints. The entailment is equivalent to satisfiability of negative constraints and can be tested in NEXPTIME [12]).

Proposition 2. *Each model $\mathcal{M}$ of the set-valued abstraction $\mathcal{P}^{\subseteq}$ of a program $\mathcal{P}$ is also a model of $\mathcal{P}$.*

Proof. If $c \equiv p(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ is a ground instance of a clause of $\mathcal{P}$, then $\{t_0\} \subseteq p \leftarrow \{t_1\} \subseteq p_1, \ldots, \{t_n\} \subseteq p_n$ is a ground instance of the corresponding implication of $\mathcal{P}^{\subseteq}$ (which holds in $\mathcal{M}$ if $\mathcal{M}$ is a model of $\mathcal{P}^{\subseteq}$, and, thus, c also holds). □

The converse of the statement above does not hold in general. Take, for example, the program $\mathcal{P}$ defined by the clause $p(f(x,x)) \leftarrow q(f(x,x))$ and the four facts $q(f(a,a)), q(f(a,b)), q(f(b,a)), q(f(b,b))$. Then $\mathcal{P}^{\subseteq}$ consists of the implication $f(x,x) \subseteq p \leftarrow f(x,x) \subseteq q$ and the four inclusions $f(a,a) \subseteq q, f(a,b) \subseteq q, f(b,a) \subseteq q, f(b,b) \subseteq q$. Then $\mathcal{M} = \{p(f(a,a))), p(f(b,b)), q(f(a,a)), q(f(a,b)), q(f(b,a)), q(f(b,b))\}$ is a model of $\mathcal{P}$ but $\mathcal{M}$ is not a model of $\mathcal{P}^{\subseteq}$. (This example transfers, in the essence, Example 1 in [1] from the setting of directional types to a general setting.) The converse of the statement in Proposition 2 does, however, hold in the special case of path closed models (Proposition 3) which we will introduce below.

The least fixpoint of $T_{\mathcal{P}^{\subseteq}}$ is in general not regular. To see this note that it is equal to the least fixpoint of $T_{\mathcal{P}}$ if, for example, $\mathcal{P}$ is the length program. This example also shows that the least model of $\mathcal{P}^{\#}$ is in general not contained in *every* model of $\mathcal{P}^{\subseteq}$. This *is* the case, however, in the special case where the model of $\mathcal{P}^{\subseteq}$ is path closed (Proposition 4).

Path closed models. The following notion of a path-closed set originates from [22]. It is equivalent to other notions occurring in the literature: tuple-distributive [27, 28], discriminative [1], or deterministic.

Definition 3 (Path-closed). *A [regular] set of ground terms is called* path-closed *if it can be defined by a [finite]* deterministic *top-down tree automaton.*

A deterministic finite tree automaton translates to a logic program which does not contain two different clauses with the same head (modulo variable renaming), e.g., $p(f(x_1, \ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n)$ and $p(f(x_1, \ldots, x_n)) \leftarrow p'_1(x_1), \ldots, p'_n(x_n)$. A discriminative set expression as defined in [1] translates to a deterministic finite tree automaton, and vice versa. That is, discriminative set expressions denote exactly path-closed regular sets. It is argued in [1] that discriminative set expressions are quite expressive and are used to express commonly used data structures. Note that lists, for example, can be defined by the program with the two clauses $list(cons(x,y)) \leftarrow list(y)$ and $list(nil)$.

The following fact is *the* fundamental property of path closed sets in the context of set constraints (see also Theorem 12 and Lemma 14 in [1]). It will be directly used in Proposition 3. For comparison, take the constraint $f(x,y) \subseteq f(a,a) \cup f(b,b)$; here, $\{f(a,a), f(b,b)\}$ is a set that is *not* path closed, and the union of the (set-valued) solutions $\theta_1 : x, y \mapsto \{a\}$ and $\theta_2 : x, y \mapsto \{b\}$ is *not* a solution. Also, take the constraint $f(x,y) \subseteq \emptyset$; here, the union of the solutions over possibly *empty* sets $\theta_1 : x \mapsto \{a\}, y \mapsto \emptyset$ and $\theta_2 : x \mapsto \emptyset, y \mapsto \{a\}$ is *not* a solution.

Lemma 1. *Solutions of conjunctions of inclusions $t \subseteq e$ between terms t interpreted over nonempty sets and expressions e denoting path closed sets of trees are closed under union; i.e., if S is a set of solutions, then Θ defined by $\Theta(x) = \bigcup\{\theta(x) \mid \theta \in S\}$ is again a solution.*

Proof. The statement follows from the fact (shown, e.g., in [13]) that in the interpretation over nonempty sets, if the upper bounds e denote path closed sets then inclusions of the form $f(x_1, \dots, x_n) \subseteq e$ are equivalent to the conjunction

$$x_1 \subseteq f_{(1)}^{-1}(e) \wedge \ldots \wedge x_n \subseteq f_{(n)}^{-1}(e)$$

where $f_{(i)}^{-1}(e)$ denotes the set $\{t_i \mid f(t_1, \dots, t_n) \in e\}$. □

The following statement underlies the *completeness* of the type checking procedure of [1] for discriminative directional types (see Theorem 12, Lemma 14 and Theorem 15 in [1]).

Proposition 3. *Each* path closed *model $\mathcal{M}$ of a program $\mathcal{P}$ is also a model of $\mathcal{P}^{\subseteq}$, its set-valued abstraction.*

Proof. Assume that the clause $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ is valid in $\mathcal{M}$ (i.e., it holds under all ground substitutions $\sigma : \mathsf{Var} \mapsto T_\Sigma$, under the interpretation of the predicates $p_0, p_1, \dots, p_n$ by $\mathcal{M}$), and that θ is a substitution mapping variables to nonempty sets such that

$$t_1\theta \subseteq p_1, \dots, t_n\theta \subseteq p_n$$

holds in $\mathcal{M}$. We have to show that also $t_0\theta \subseteq p_0$ holds in $\mathcal{M}$. The assumption yields that, for every ground substitution $\sigma : \mathsf{Var} \mapsto T_\Sigma$ such that $\sigma(x) \in \theta(x)$ for all $x \in \mathsf{Var}$,

$$t_1\sigma \in p_1, \dots, t_n\sigma \in p_n$$

holds in $\mathcal{M}$. Thus, also $t_0\sigma \in p_0$ holds in $\mathcal{M}$. Since we have that

- $t_0\sigma \in p_0$ is equivalent to $t_0\bar{\sigma} \subseteq p_0$ where $\bar{\sigma}$ is the set substitution defined by $\bar{\sigma}(x) = \{\sigma(x)\}$,
- θ is the union of all $\bar{\sigma}$ such that $\sigma(x) \in \theta(x)$ for all $x \in \mathsf{Var}$,
- solutions of $t_0 \subseteq p_0$, where p_0 is interpreted by $\mathcal{M}$ as a path closed set, are closed under union (Lemma 1),

the inclusion $t_0 \subseteq p_0$ also holds in $\mathcal{M}$ under the substitution θ. □

Path closure abstraction. The path closure PC of a set M of trees is the smallest path closed set containing M. We consider a third abstraction of the operator $T_\mathcal{P}$ by composing the path closure PC with $T_\mathcal{P}$. We note that we do not know whether the least fixpoint of the operator $PC \circ T_\mathcal{P}$ is always a regular set. The following comparison of two of the three abstractions that we have defined so far will be used in the proof of Proposition 4.

Remark 3 (Set-based approximation and path closure). The path closure abstraction of the direct-consequence operator of $\mathcal{P}$ approximates also its set-based abstraction; i.e.,

$$T_{\mathcal{P}^{\#}} \leq (PC \circ T_{\mathcal{P}}).$$

Proof. We use the equality $T_{\mathcal{P}^{\#}} = \tau_{\mathcal{P}}$. If $p_0(t_0[x_1\theta_1, \ldots, x_m\theta_m]) \in \tau_{\mathcal{P}}(M)$ because $p_1(t_1\theta_j), \ldots, p_n(t_n\theta_j) \in M$, then $p_0(t_0\theta_j) \in T_{\mathcal{P}}(M)$, for $j = 1, \ldots, m$. But then we have $p_0(t_0[x_1\theta_1, \ldots, x_m\theta_m]) \in (PC \circ T_{\mathcal{P}})(M)$. $\square$

We will use the following statement later in order to compare the directional types obtained by our type inference procedure with the subclass of discriminative directional types for which the type check in [1] is sound and complete. (Note that the path closure of a model of a program is in general not itself a model, and that the least model of $\mathcal{P}^{\#}$ is in general not contained in the path closure of $\mathcal{P}$).

Proposition 4. *The least model of $\mathcal{P}^{\#}$, the set-based abstraction of a program $\mathcal{P}$, is contained in every path closed model of $\mathcal{P}^{\subseteq}$, the set-valued abstraction of a program $\mathcal{P}$.*

Proof. By Remark 3, $T_{\mathcal{P}^{\#}} \subseteq (PC \circ T_{\mathcal{P}})$ and thus, by Remark 2, $T_{\mathcal{P}^{\#}} \subseteq (PC \circ T_{\mathcal{P}^{\subseteq}})$. If $\mathcal{M}$ is a path-closed model of $\mathcal{P}^{\subseteq}$, then it is also a fixpoint of $PC \circ T_{\mathcal{P}}$, and hence it contains the least fixpoint of $T_{\mathcal{P}^{\#}}$, i.e., the least model of $\mathcal{P}^{\#}$. $\square$

3 Directional Types and Type Programs

A *type* T is a set of terms t closed under substitution [2]. A *ground type* is a set of ground terms (i.e., trees), and thus a special case of a type. A term t has type T, in symbols $t:T$, if $t \in T$. A *type judgment* is an implication $t_1:T_1 \wedge \ldots \wedge t_n:T_n \rightarrow t_0:T_0$ that holds under all term substitutions $\theta : \mathsf{Var} \rightarrow T_{\Sigma}(\mathsf{Var})$.

Definition 4 (Directional type of a program [8, 1]). *A* directional type *of a program $\mathcal{P}$ is a family $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \mathsf{Pred}}$ assigning to each predicate p of $\mathcal{P}$ an input type I_p and an output type O_p such that, for each clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ of $\mathcal{P}$, the following type judgments hold.*

$$\begin{aligned}
t_0:I_{p_0} &\rightarrow t_1:I_{p_1} \\
t_0:I_{p_0} \wedge t_1:O_{p_1} &\rightarrow t_2:I_{p_2} \\
&\vdots \\
t_0:I_{p_0} \wedge t_1:O_{p_1} \wedge \ldots \wedge t_{n-1}:O_{p_{n-1}} &\rightarrow t_n:I_{p_n} \\
t_0:I_{p_0} \wedge t_1:O_{p_1} \wedge \ldots \wedge t_n:O_{p_n} &\rightarrow t_0:O_{p_0}
\end{aligned}$$

We then also say that $\mathcal{P}$ is well-typed wrt. $\mathcal{T}$. *A program together with its query $main(t)$ is well-typed wrt. $\mathcal{T}$ if furthermore the query argument t is well-typed wrt. the input type for main (i.e, the type judgment $t:I_{main}$ holds).*

Definition 5 (Ordering on directional types). *We define that* $\mathcal{T} = (I_p \to O_p)_{p\in\mathsf{Pred}}$ *is* uniformly more precise *than* $\mathcal{T}' = (I'_p \to O'_p)_{p\in\mathsf{Pred}}$ *if* $I_p \subseteq I'_p$ *and* $O_p \subseteq O'_p$ *for all predicates* p.

The least precise directional type for which any program (possibly together with a query) is well-typed is $\mathcal{T}_\top = (\top \to \top)_{p\in\mathsf{Pred}}$ assigning the set of all terms to each input and output type. In the absence of a query, the most precise one is $\mathcal{T}_\perp = (\perp \to \perp)_{p\in\mathsf{Pred}}$ assigning the empty set to each input and output type. This changes if, for example, a query of the form *main* is present (see Section 4).

Definition 6 ($Sat(T)$, the type of terms satisfying T [1]). *Given the ground type* T, *the set* $Sat(T)$ *of terms satisfying* T *is the type*

$$Sat(T) = \{t \in T_\Sigma(\mathsf{Var}) \mid \theta(t) \in T \textit{ for all ground substitutions } \theta : \mathsf{Var} \to T_\Sigma\}.$$

Definition 7 (Discriminative type). *A directional type is called* discriminative *if it is of the form* $(Sat(I_p) \to Sat(O_p))_{p\in\mathsf{Pred}}$, *where the sets* I_p, O_p *are path-closed.*

Remark 4. The clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ is valid in some model $\mathcal{M}$ if and only if the type judgment

$$t_0 : Sat(p_0) \leftarrow t_1 : Sat(p_1) \wedge \ldots \wedge t_n : Sat(p_n)$$

holds in $\mathcal{M}$ (i.e., under the interpretation of $p_0, p_1, \ldots, p_n$ by $\mathcal{M}$).

Proof. Membership of the application of substitutions to terms in sets of the form $Sat(E)$ is defined by the application of ground substitutions to the terms in E. $\square$

A directional type of the form $\mathcal{T} = (Sat(I_p) \to Sat(O_p))_{p\in\mathsf{Pred}}$, for ground types $I_p, O_p \subseteq T_\Sigma$, satisfies a type judgment if and only if the corresponding directional *ground* type $\mathcal{T}_g = (I_p \to O_p)_{p\in\mathsf{Pred}}$ does.

We will next transform the well-typedness condition in Definition 4 into a logic program by replacing $t : I_p$ with the atom $p^{In}(t)$ and $t : O_p$ with $p^{Out}(t)$.

Definition 8 ($\mathcal{P}_{InOut}$, the type program for $\mathcal{P}$). *Given a program* $\mathcal{P}$, *the corresponding type program* $\mathcal{P}_{InOut}$ *defines an in-predicate* p^{In} *and an out-predicate* p^{Out} *for each predicate* p *of* $\mathcal{P}$. *Namely, for every clause* $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ *in* $\mathcal{P}$, $\mathcal{P}_{InOut}$ *contains the* n *clauses defining in-predicates corresponding to each atom in the body of the clause,*

$$\begin{aligned}
&p_1^{In}(t_1) \leftarrow p_0^{In}(t_0)\\
&p_2^{In}(t_2) \leftarrow p_0^{In}(t_0),\ p_1^{Out}(t_1)\\
&\qquad\vdots\\
&p_n^{In}(t_n) \leftarrow p_0^{In}(t_0),\ p_1^{Out}(t_1), \ldots,\ p_{n-1}^{Out}(t_{n-1})
\end{aligned}$$

and the clause defining the out-predicate corresponding to the head of the clause,

$$p_0^{Out}(t_0) \leftarrow p_0^{In}(t_0),\ p_1^{Out}(t_1), \ldots,\ p_n^{Out}(t_n).$$

If the program $\mathcal{P}$ comes together with a query $main(t)$, we add the clause $main^{In}(t) \leftarrow true$ to $\mathcal{P}_{InOut}$. The next statement extends naturally to a characterization of the well-typedness of a program together with a query.

Theorem 1 (Types and models of type programs). *The program $\mathcal{P}$ is well-typed wrt. the directional type*

$$\mathcal{T} = (Sat(I_p) \to Sat(O_p))_{p \in \mathsf{Pred}}$$

(with ground types I_p, O_p) if and only if the subset of the Herbrand base corresponding to $\mathcal{T}$,

$$\mathcal{M}_\mathcal{T} = \{p^{In}(t) \mid t \in I_p\} \cup \{p^{Out}(t) \mid t \in O_p\},$$

is a model of the type program $\mathcal{P}_{InOut}$.

Proof. The validity of the well-typing conditions under *ground* substitutions is exactly the logical validity of the clauses of $\mathcal{P}_{InOut}$ in the model $\mathcal{M}_\mathcal{T}$. The statement then follows by Remark 4. □

We next define two abstractions of type programs. We will use $\mathcal{P}^{\#}_{InOut}$ for type inference (Section 4) and $\mathcal{P}^{\subseteq}_{InOut}$ for type checking (Section 5). Given a directional type $\mathcal{T}$, the interpretation of $\mathcal{P}^{\subseteq}_{InOut}$ by the corresponding subset $\mathcal{M}_\mathcal{T}$ is the *set constraint condition* which is used in [1] to replace the well-typedness condition in Definition 4.

Definition 9 (Abstractions of type programs). *The set*-based *type program $\mathcal{P}^{\#}_{InOut}$ is the set-based abstraction of $\mathcal{P}_{InOut}$, and the set*-valued *type program $\mathcal{P}^{\subseteq}_{InOut}$ is the set-valued abstraction of $\mathcal{P}_{InOut}$; i.e.,*

$$\mathcal{P}^{\#}_{InOut} = (\mathcal{P}_{InOut})^{\#},$$
$$\mathcal{P}^{\subseteq}_{InOut} = (\mathcal{P}_{InOut})^{\subseteq}.$$

The following direct consequence of Theorem 1 and Propositions 2 and 3 restates the soundness and the conditional completeness of the type check in [1].

Theorem 2 ([Discriminative] types and models of set-valued type programs). *A directional type of the form $\mathcal{T} = (Sat(I_p) \to Sat(O_p))_{p \in \mathsf{Pred}}$ is a directional type of the program $\mathcal{P}$ if the corresponding subset $\mathcal{M}_\mathcal{T}$ of the Herbrand base is a model of $\mathcal{P}^{\subseteq}_{InOut}$, the set-valued abstraction of the type program of $\mathcal{P}$. For* discriminative *directional types $\mathcal{T}$, the converse also holds.*

Proof. The first part follows from Proposition 2 together with Theorem 1, the second from from Proposition 3 together with Theorem 1. □

4 Directional Type Inference

We consider three different scenarios in which we may want to infer directional types from a program.

(1) The program comes together with a query consisting of one atom $main$ without arguments (and there is a clause $main \leftarrow p_1(t_1), \dots, p_n(t_n)$ calling the actual query). In this case, we are interested in inferring the most precise directional type $\mathcal{T}$ such that $\mathcal{P}$ together with the query $main$ is well-typed.

According to the model-theoretic characterization of the well-typedness of a program together with a query (Theorem 1), we must add the clause $main^{In} \leftarrow true$ to $\mathcal{P}_{InOut}$. This means that $\mathcal{T}_\perp = (\perp \rightarrow \perp)_{p \in \mathsf{Pred}}$ is generally not a directional type of a program together with the query $main$ and, hence, it is nontrivial to infer a precise one.

(2) The program comes together with a query consisting of one atom $main(t)$ and a lower bound M_{main} for the input type of $main$ is given (the user hereby encodes which input terms to the query predicates are expected). In this case, we are interested in inferring the most precise directional type $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \mathsf{Pred}}$ for $\mathcal{P}$ such that the input type for $main$ lies above the lower bound for $main$, i.e., such that $M_{main} \subseteq I_{main}$.

For example, take the program defining the predicate $reverse(x, y)$

$$reverse([], []).$$
$$reverse([X|Xs], Y) \leftarrow reverse(Xs, Z), append(Z, [X], Y,).$$

together with the definition of $append$ and with the query $reverse(x, y)$. If the expected input terms are lists (for x) and non-instantiated variables (for y), i.e., the lower bound specified is $M_{rev} = (list, \top)$, then the type inferred by our algorithm for $reverse$ is $(list, \top) \rightarrow (list, list)$ and the type inferred by our algorithm for $append$ is $(list, [\top], \top) \rightarrow (list, [\top], list)$, where $[\top]$ is the type of all singleton lists.

(3) Lower bounds M_p for the input types I_p of *all* predicates p of $\mathcal{P}$ are given. This may be done explicitly for some p and implicitly, with $M_p = \emptyset$, for the others. In this case, we are interested in inferring the most precise directional type $\mathcal{T} = (I_p \rightarrow O_p)_{p \in \mathsf{Pred}}$ for $\mathcal{P}$ such that the input types for p lie above the given lower bounds, i.e., such that $M_p \subseteq I_p$ for all p.

In a setting with program modules, for example, the lower bounds that are explicitly specified may be the query goals of exported predicates.

The scenario (3) resembles the one imagined by Aiken and Lakshman in the conclusion of [1]. Note that in our setting, however, the sets M_p need not already be input types. For example, if the inputs for x and y in $append(x, y, z)$ are expected to be lists of even length only, i.e., the lower bound for I_{app} is given by $M_{app} = (evenlist, evenlist, \top)$, then we would infer the set of all lists as the input type for x, i.e., $I_{app} = (list, evenlist, \top)$ (note that there are recursive calls to append with lists of odd length).

We obtain (1) and (2)as the special case of (3) where the lower bounds for all input types but $main$ are given as the empty set. Hence, it is sufficient to formulate the type inference only for the case (3).

Definition 10 (Inference of the set-based directional type $\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p\in\mathsf{Pred}})$). *Given a program $\mathcal{P}$ and a family of lower bounds M_p for the input types of the predicates p of the program, we infer the set-based directional type*

$$\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p\in\mathsf{Pred}}) = (Sat(I_p) \to Sat(O_p))_{p\in\mathsf{Pred}}$$

where I_p and O_p are the denotations of the predicates p^{In} and p^{Out} in the least model of the program

$$\mathcal{P}^{\#}_{InOut} \;\cup\; \{p^{In}(x) \leftarrow M_p(x) \mid p \in \mathsf{Pred}\}. \tag{2}$$

The definition of type inference above leaves open in which formalism the lower bounds are specified and how the inferred directional types are presented to the user. There is a wide variety of formalisms that coincide in the expressive power of regular sets of trees and that can be effectively translated one into another and, hence, for which our type inference yields an effective procedure.

More concretely, we propose to represent the lower bounds M_p through logic programs in restricted syntax (see Section 2) that corresponds directly to alternating tree automata (and also to the positive set expressions considered in [1]; an even more restricted syntax corresponds to non-deterministic tree automata and to positive set expressions without intersection). We attach these logic programs to the program in (2) as definitions of the sets M_p. Then, we can apply one of the known algorithms (see, e.g., [19, 18, 11, 10]) in order to compute, in single-exponential time (in the size of the program $\mathcal{P}$ and the programs for the sets M_p), a logic program that corresponds to a non-deterministic tree automaton and that is equivalent to the program in (2) wrt. the least model and, thus, represents $\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p\in\mathsf{Pred}})$. The representation of sets by a non-deterministic tree automaton is a good representation in the sense that, for example, the test of emptiness can be done in linear time.

If, in Definition 10, we replace $\mathcal{P}^{\#}_{InOut}$ with $\mathcal{P}_{InOut}$, then we obtain the uniformly most precise directional type $\mathcal{T}_{min}(\mathcal{P})$ wrt. given lower bounds for input types. In general, we cannot effectively compute $\mathcal{T}_{min}(\mathcal{P})$ (e.g., test emptiness of its input and output types).

We repeat that the following comparison of the set-based directional type of $\mathcal{P}$ with discriminative directional types of $\mathcal{P}$ is interesting because these form the subclass of directional types for which the type check in [1] is sound and complete.

Theorem 3 (Soundness and Quality of Type Inference). *The program $\mathcal{P}$ is well-typed wrt. the set-based directional type $\mathcal{T}_{sb}(\mathcal{P}, (M_p)_{p\in\mathsf{Pred}})$. Moreover, this type is uniformly more precise than every discriminative directional type of $\mathcal{P}$ whose family of input types contains $(M_p)_{p\in\mathsf{Pred}}$.*

Proof. The two statements are direct consequences of Propositions 1 and 4, respectively, together with Theorem 1. □

5 Complexity of Directional Type Checking

Theorem 4. *The complexity of directional type checking of logic programs for discriminative types is* DEXPTIME-complete.

Proof. The DEXPTIME-hardness follows from refining the argument in the proof of Theorem 18 in [1] with the two facts that (1) discriminative types are denoted by path closed sets, and (2) testing the non-emptiness of a sequence of n tree automata is DEXPTIME-hard even if the automata are restricted to deterministic ones (which recognize exactly path closed sets) [29].

We obtain a type checking algorithm in single-exponential time as follows. The input is the program $\mathcal{P}$ and the discriminative directional type

$$\mathcal{T} = (Sat(I_p) \rightarrow Sat(O_p))$$

where all types I_p and O_p are given by ground set expressions (a special case of which are regular tree expressions or non-deterministic tree automata). Ground set expressions correspond to alternating tree automata which are self-dual; i.e., we can obtain ground set expressions $\widetilde{I_p}$ and $\widetilde{O_p}$ representing the complement by a syntactic transformation in linear time. We can translate ground expressions into logic programs defining predicates p_{I_p} and p_{O_p} such that they denote I_p and O_p wrt. the least-model semantics (and similarly predicates $p_{\widetilde{I_p}}$ and $p_{\widetilde{O_p}}$ for $\widetilde{I_p}$ and $\widetilde{O_p}$).

We now use one of the well-known single-exponential time algorithms [19, 18, 11, 10] to compute (the non-deterministic tree automaton representing) the least model of the set-based abstraction $\mathcal{P}^{\#}$ of a logic program $\mathcal{P}$. We apply such an algorithm to the program $\mathcal{P}^{\#}_{InOut}(\mathcal{T})$ that we obtain from $\mathcal{P}^{\#}_{InOut}$ by adding the clauses $p^{In}(x) \leftarrow p_{I_p}(x)$ and $p^{Out}(x) \leftarrow p_{O_p}(x)$ and the logic programs defining p_{I_p} and p_{O_p}.

We use the result in order to test whether the denotations of p^{In} and p^{Out} under the least model of the program $\mathcal{P}_{InOut}(\mathcal{T})$ are exactly I_p and O_p. This holds if and only if $\mathcal{T}$ is a directional type of $\mathcal{P}$ (otherwise, we have a proper inclusion for at least one p; see the correctness proof below). The test works by testing whether the intersection of (the non-deterministic tree automaton representing) the denotations of p^{In} and p^{Out} under the least model of the program $\mathcal{P}_{InOut}(\mathcal{T})$ with the complements $p_{\widetilde{I_p}}$ and $p_{\widetilde{O_p}}$ is empty. This can be done in one pass by taking the conjunction of $\mathcal{P}_{InOut}(\mathcal{T})$ with the logic programs defining $p_{\widetilde{I_p}}$ and $p_{\widetilde{O_p}}$ and testing the emptiness of new predicates defined as the intersection of p_{I_p} and $p_{\widetilde{I_p}}$ (and p_{O_p} and $p_{\widetilde{O_p}}$).

The correctness of the algorithm follows with Theorem 1 and Proposition 1 and 4. In detail: Given a discriminative directional type $\mathcal{T}$ and a program $\mathcal{P}$, we have that $\mathcal{P}$ is well-typed wrt. $\mathcal{T}$ iff the corresponding subset $\mathcal{M}_{\mathcal{T}}$ of the Herbrand base is a (path closed) model of $\mathcal{P}_{InOut}$ by Theorem 1. If this is the case then $\mathcal{M}_{\mathcal{T}}$ is equal to the least model $\mathcal{M}_0$ of $\mathcal{P}_{InOut}(\mathcal{T})$, since $\mathcal{M}_{\mathcal{T}} \subseteq \mathcal{M}_0$ by definition of $\mathcal{P}_{InOut}(\mathcal{T})$, and $\mathcal{M}_{\mathcal{T}} \supseteq \mathcal{M}_0$ holds by Proposition 4 (note that $\mathcal{M}_{\mathcal{T}}$

is a path closed model of $\mathcal{P}_{InOut}$ and of the additional clauses translating $\mathcal{T}$ and, thus, contains the least one). On the other hand, if $\mathcal{M}_{\mathcal{T}}$ is equal to the least model of $\mathcal{P}^{\#}_{InOut}(\mathcal{T})$, then $\mathcal{T}$ is a directional type by Proposition 1 and Theorem 1. □

We note that the procedure above yields a semi-test (in the same sense as the one in [1]) for well-typedness wrt. the class of *general* directional types, since the equivalence between the inferred type and the given one is a sufficient (but generally not necessary) condition for well-typedness wrt. the given type. We repeat that it implements a full test wrt. discriminative directional types.

6 Future work

One of the obvious directions for future work is an implementation. We already have a working prototype implementation on top of the saturation-based theorem prover SPASS [30]. The first results are very promising; due to specific theorem-proving techniques like powerful redundancy criteria, one obtains a decision procedure for the emptiness test that is quite efficient on our examples. We ran our prototype on the benchmark programs from [9]. We were able to detect emptiness of the main predicate for six of the twelve programs, while a similar method of Gallagher succeeded in three cases. We believe that by using the tree-automata techniques suggested in [18] together with automata minimization (and conversion to the syntax of ground set expressions), we can further improve the efficiency of our implementation and the readability of the output.

Acknowledgments

We thank David McAllester for turning us on to magic sets and thereby to directional types. We thank Harald Ganzinger for useful comments.

References

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In B. L. Charlier, editor, *1st International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 43–60, Namur, Belgium, Sept. 1994. Springer Verlag.
2. K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 12–35, Vancouver, Canada, 1993. The MIT Press.
3. K. R. Apt. Program verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems*, pages 55–95. Oxford University Press, 1995.
4. K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of *lncs*, pages 1–19, Gdansk, Poland, 30 Aug.– 3 Sept. 1993. Springer.

5. J. Boye. *Directional Types in Logic Programming.* PhD thesis, Department of Computer and Information Science, Linköping University, 1996.
6. J. Boye and J. Maluszynski. Two aspects of directional types. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 747–764, Cambridge, June13–18 1995. MIT Press.
7. J. Boye and J. Maluszynski. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, Dec. 1997.
8. F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335, Washington, USA, 1992. The MIT Press.
9. M. Bruynooghe, H. Vandecasteele, A. de Waal, and M. Denecker. Detecting unsolvable queries for definite logic programs. In *Proceedings of the Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, LNCS. Springer-Verlag, 1998. to appear.
10. W. Charatonik, D. McAllester, D. Niwiński, A. Podelski, and I. Walukiewicz. The Horn mu-calculus. To appear in Vaughan Pratt, editor, Proceedings of the 13th IEEE Annual Symposium on Logic in Computer Science.
11. W. Charatonik, D. McAllester, and A. Podelski. Computing the least and the greatest model of the set-based abstraction of logic programs. Presented at the Dagstuhl Workshop on Tree Automata, October 1997.
12. W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 128–136, 1994.
13. W. Charatonik and A. Podelski. Set constraints for greatest models. Technical Report MPI-I-97-2-004, Max-Planck-Institut für Informatik, April 1997. www.mpi-sb.mpg.de/~podelski/papers/greatest.html.
14. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In B. L. Charlier, editor, *Proceedings of the First International Static Analysis Symposium*, Lecture Notes in Computer Science 864, pages 281–296. Springer Verlag, 1994.
15. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *LNCS*, pages 22–50. Springer-Verlag, June 1996.
16. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.
17. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Record of FPCA '95 - Conference on Functional Programming and Computer Architecture*, pages 170–181, La Jolla, California, USA, 25-28 June 1995. SIGPLAN/SIGARCH/WG2.8, ACM Press, New York, USA.
18. P. Devienne, J.-M. Talbot, and S. Tison. Set-based analysis for logic programming and tree automata. In *Proceedings of the Static Analysis Symposium, SAS'97*, volume 1302 of *LNCS*, pages 127–140. Springer-Verlag, 1997.
19. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.
20. J. Gallagher and D. A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.

21. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. V. Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margherita Ligure, Italy, 1994. The MIT Press.
22. F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
23. N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779, Washington, USA, 1992. The MIT Press.
24. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.
25. N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 281–298. Springer-Verlag, 1994.
26. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
27. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
28. Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 85–97, Washington, USA, 1992. The MIT Press.
29. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52:57–60, 1994.
30. C. Weidenbach. Spass version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, 1997.
31. E. Yardeni and E. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog*, volume 2, chapter 28, pages 211–244. The MIT Press, 1987.