Directional Type Checking for Logic Programs: Beyond Discriminative Types

Witold Charatonik

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
`www.mpi-sb.mpg.de/~witold`
and
University of Wrocław, Poland

Abstract. 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. It is known that directional-type checking wrt. arbitrary types is undecidable; several authors proved decidability of the problem wrt. discriminative regular types. In this paper, using techniques based on tree automata, we show that directional-type checking for logic programs wrt. general regular types is DEXPTIME-complete and fixed-parameter linear. The latter result shows that despite the exponential lower bound, the type system might be usable in practice.
Keywords: types in logic programming, directional types, regular types, tree automata.

1 Introduction

It is commonly agreed that types are useful in programming languages. They help understanding programs, detecting errors or automatically performing various optimizations. Although most logic programming systems are untyped, a lot of research on types in logic programming has been done [33].

Regular types. Probably the most popular approach to types in logic programming uses *regular* types, which are sets of ground terms recognized by finite tree automata (in several papers, including this one, this notion is extended to non-ground terms). Intuitively, regular sets are finitely representable sets of terms, just as in case of regular sets of words, which are finitely representable by finite word automata.

Actually, almost all type systems occurring in the literature are based on some kinds of regular grammars which give a very natural (if not the only) way to effectively represent interesting infinite collections of terms that denote e.g. lists or other recursive data structures. Of course some of them use extensions of regular sets with non-regular domains like numbers (see the discussion in Section 3.3), nonground terms (where all types restricted to ground terms are regular), polymorphism (where all monotypes, that is ground instances of polymorphic types, are regular). Very few systems go further beyond regular sets (and usually say nothing about computability issues for combining types).

Prescriptive and descriptive approaches. There are two main streams in the research on types in logic programming. In the prescriptive stream the user has to provide type declarations for predicates; these declarations form an integral part of the program. The system then checks if the program is well-typed, that is, if the type declarations are consistent. The present paper falls into the prescriptive stream.

In the descriptive stream the types are inferred by the system and used to describe semantic properties of untyped programs. The basic idea here is to over-approximate the least model of a given program by a regular set. This approach can be found in particular in [30, 39, 25, 19, 16, 26, 21, 17], or, using a type-graph representation of regular sets (a type graph may be seen as a deterministic top-down tree automaton), in [29, 28]. An advantage of the descriptive approach is that there is no need for type declarations; a disadvantage is that the inferred types may not correspond to the intent of the programmer.

The approximation of the least model of the program by a regular set is often not as precise as one would expect. A typical example here is a clause $append([], L, L)$. Most of the systems approximate the success set of this clause by the set of triples $\langle [], x, y\rangle$ where x and y are any terms and thus loose the information that a second and third argument are of the same type. To overcome this problem, [24, 20] introduced approximations based on magic-set transformation of the input program. It was observed in [12] that types of the magic-set transformation of a program coincide with directional types of the initial program as they appear in [35, 8, 4, 2, 1, 3, 6, 5, 7].

Directional types. 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. They first occurred in [35] as predicate profiles and in [8] as mode dependencies. Our use of the terminology "directional type" stems from [1].

Discriminative types. In most type systems for logic programs that are based on regular types, the types are restricted to be *discriminative* (equivalently, path-closed or tuple-distributive or recognizable by deterministic top-down tree automata). The reason for that is probably a hope for better efficiency or conceptual simplicity of such approach. Unfortunately, discriminative sets are closed under neither union nor complementation. A union of two discriminative sets is then approximated by a least discriminative set that contains both of them, but then the distributivity laws for union and intersection do not hold anymore. This is very unintuitive and has lead already to several wrong results. One of the results of this paper is that in the context of directional types the restriction to discriminative types, at least theoretically, does not pay: the exponential lower bound for the discriminative case is matched by the exponential upper bound for the general case. In fact, as shown in [14], even stronger restriction to unary types (where all paths in a tree are disconnected from each other, see e.g. [39]) is not worth it, since type checking problem (even for non-directional types) remains hard for DEXPTIME.

Complexity of type checking. The exponential lower bound of the type-checking problem looks quite discouraging at the first view. A closer look into the proof of the lower bound shows that the program used there was very short while the types were quite large (the encoding of the Turing-machine computation was done in the types, not in the program). The situation in practice looks usually quite opposite: the type-checking problems are usually applied to large programs and rather small types. This lead us to study the parameterized complexity of the problem, where the parameter is given by the length of the type to be checked. We obtained here quite a nice result: the problem is fixed-parameter linear, which means that for a fixed family of types the type-checking problem can be decided in time linear in the size of the program. This shows that there is a good potential for the type system to be practical. A similar phenomenon is known already for the functional language ML where bad theoretical lower bounds do not match good practical behaviour of the time system. The explanation was given by Henglein [27] who showed that typability by types of size bounded by constant is polynomial time decidable.

Related work. It is pointed out in [1] that the type checking problem is undecidable for arbitrary directional types. Therefore Aiken and Lakshman restrict themselves to regular directional types. Although their algorithm for automatic type checking is sound for general regular types, it is sound and complete only for discriminative ones. It is based on solving negative set constraints and thus runs in nondeterministic exponential time. Another algorithm (without complexity analysis) for type-checking for discriminative directional types is given in [5]. In [12] it is proved that directional type checking wrt. discriminative types is DEXPTIME-complete and an algorithm for *inferring* (regular, not necessarily discriminative) directional types is given.

Rychlikowski and Truderung [36] proposed recently a system of polymorphic directional types. The types there are incomparable with ours: on one hand they are more general because of the use of the polymorphism; on the other hand they are even more restricted than regular discriminative types (e.g. they are not able to express lists of an even length). The authors presented a type-checking algorithm working in DEXPTIME, but probably the most interesting feature of this system is the inference of so-called main type of a predicate — the type that provides a compact representation of all types of the predicate.

Our results. The methods used in the mentioned papers are not strong enough to prove the decidability of directional type checking wrt. general regular types. In this paper, using tree-automata techniques, we prove that this problem is decidable in DEXPTIME, which, together with the result from [12] stating DEXPTIME-hardness, establishes DEXPTIME-completeness of the problem. Moreover, we show that the problem is fixed-parameter linear – our procedure is exponential in the size of the input types, but linear in the size of the program. This improves the results by Aiken and Lakshman [1], Boye [5], and Charatonik and Podelski [12], where decidability is restricted to discriminative types.

Decidability of directional type checking wrt. general regular types has already a quite long history. It was first proved [11] by a reduction to the encompassment theory [9]. The result was not satisfactory because of the complexity of the obtained procedure: several (around five) times exponential. In [10] we found another solution based on a different kind of automata and reduced the complexity to NEXPTIME. The proof presented here is a refinement of the argument from [10].

2 Preliminaries

If Σ is a signature (that is, set of function symbols) and Var is a set of variables then T_Σ is the set of ground terms and $T_{\Sigma(\mathsf{Var})}$ is the set of non-ground terms over Σ and Var. We write $\mathsf{Var}(t)$ for the set of variables occurring in the term t. The notation $|S|$ is used, depending on the context, for the cardinality of the set S or for the size of the object S (that is, the length of the word encoding S).

2.1 Tree automata

The methods we use are based on tree-automata techniques. Standard techniques as well as all well-known results that we mention here can be found e.g. in [22, 15, 38]. Below we recall basic notions in this area.

Definition 1 (Tree automaton). *A tree automaton is a tuple* $\mathcal{A} = \langle \Sigma, Q, \Delta, F\rangle$ *where* Σ, Q, Δ, F *are finite sets such that*

- Σ *is a signature,*
- Q *is a finite set of states,*
- Δ *is set of transitions of the form* $f(q_1, \ldots, q_n) \to q$ *where* $f \in \Sigma$, $q, q_1, \ldots, q_n \in Q$ *and* n *is the arity of* f,
- $F \subseteq Q$ *is a set of final states.*

The automaton $\mathcal{A}$ *is called*

- *bottom-up deterministic, if for all* $f \in \Sigma$ *and all sequences* $q_1, \ldots, q_n \in Q$ *there exists at most one* $q \in Q$ *such that* $f(q_1, \ldots, q_n) \to q \in \Delta$,
- *top-down*[1] *deterministic if* $|F| = 1$ *and for all* $f \in \Sigma$ *and all* $q \in Q$ *there exists at most one sequence* $q_1, \ldots, q_n \in Q$ *such that* $f(q_1, \ldots, q_n) \to q \in \Delta$,
- *complete, if for all* $f \in \Sigma$ *and all sequences* $q_1, \ldots, q_n \in Q$ *there exists at least one* $q \in Q$ *such that* $f(q_1, \ldots, q_n) \to q \in \Delta$.

A tree automaton $\mathcal{A} = \langle \Sigma, Q, \Delta, F\rangle$ translates to a logic program containing a clause $q(f(x_1, \ldots, x_n)) \leftarrow q_1(x_1), \ldots, q_n(x_n)$ for each transition $f(q_1, \ldots, q_n) \to q \in \Delta$, where one is interested only in queries about the predicates in F.

Definition 2 (Run). *A run of a tree automaton* $\mathcal{A} = \langle \Sigma, Q, \Delta, F\rangle$ *on a tree* $t \in T_\Sigma$ *is a mapping* ρ *assigning a state to each occurrence of a subterm* $f(t_1, \ldots, t_n)$ *of* t *such that*

$$f(\rho(t_1), \ldots, \rho(t_n)) \to \rho(f(t_1, \ldots, t_n)) \in \Delta.$$

A run ρ *on* t *is* successful *if* $\rho(t) \in F$.

[1] Intuitively, a top-down automaton reads trees top-down, and thus F is here the set of initial (not final) states.

Sometimes we will refer to runs over terms in $T_{\Sigma \cup Q}$. We then extend the definition above by putting $\rho(q) = q$ for all states in Q.

If there exists a successful run on a tree t then we say that the automaton accepts, or recognizes, t. The set of all trees accepted by an automaton $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is called the *language* of the automaton $\mathcal{A}$, or the set *recognized* by this automaton. A set of trees is called *regular* if it is recognized by some tree automaton.

A state q of the automaton $\mathcal{A}$ is called [bottom-up] *reachable* if there exists a tree $t \in T_\Sigma$ and a run ρ of $\mathcal{A}$ on t such that $\rho(t) = q$.

It is well-known (cf. [22, 15, 38]) that regular languages are closed under Boolean operations: one can effectively construct in polynomial time an automaton that recognizes union or intersection of given two regular languages, and in exponential time an automaton that recognizes complement. In polynomial time one can compute the set of reachable states and thus test emptiness of the language recognized by a given automaton. In exponential time one can determinize an automaton, that is, construct a bottom-up deterministic automaton that recognizes the same set. Tree automata are not top-down determinizable.

Example 1. Consider the automaton $\mathcal{A} = \langle \{a, f\}, \{q_0, q_1, q\}, \{a \to q_0, a \to q_1, f(q_0, q_1) \to q\}, \{q\} \rangle$. The run that assigns q_0 to the first occurrence of a, q_1 to the second occurrence of a and q to $f(a, a)$ is a successful run of $\mathcal{A}$ on $f(a, a)$. The automaton $\mathcal{A}$ is top-down deterministic, is not bottom-up deterministic, and is not complete.

2.2 Directional types

By logic programs we mean definite horn-clause programs (pure Prolog programs). For the sake of simplicity we assume that all predicate symbols occurring in this paper are unary (there is no loss of generality since function symbols may be used to form tuples). The set of predicate symbols occurring in a program $\mathcal{P}$ is denoted $\mathsf{Pred}(\mathcal{P})$ or simply Pred if $\mathcal{P}$ is clear from the context. For a program $\mathcal{P}$, $lm(\mathcal{P})$ denotes its least model. For $p \in \mathsf{Pred}(\mathcal{P})$ we define $[\![p]\!]_\mathcal{P} = \{t \mid p(t) \in lm(\mathcal{P})\}$.

A *type* is a set of terms 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 $t_1{:}T_1 \wedge \ldots \wedge t_n{:}T_n \to t_0{:}T_0$. We say that such a judgment *holds* if the implication $t_1\theta \in T_1 \wedge \ldots \wedge t_n\theta \in T_n \to t_0\theta \in T_0$ is true for all term substitutions $\theta : \mathsf{Var} \to T_{\Sigma(\mathsf{Var})}$.

We recall that a set of ground terms is regular if it can be defined by a finite tree automaton (or, equivalently, by a ground set expression as in [1] or a regular grammar as in [16]). The definition below coincides with the types used in [1], it extends the definition from [16] by allowing non-ground types, and is equivalent to the definition from [5].

Definition 3 (Regular type). *A type is* regular *if it is of the form $Sat(T)$ for a regular set T of ground terms, where the set $Sat(T)$ of terms satisfying T is* *the type*

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

Definition 4 (Directional type of a program [8, 1]). *A* directional type *of a program $\mathcal{P}$ is a family*

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

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.

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

We then also say that $\mathcal{P}$ is well-typed wrt. $\mathcal{T}$.

Following [1] we define that a query $q_1(t_1), \ldots, q_n(t_n)$ is well-typed if for all $1 \le j \le n$ the judgment $\bigwedge_{1 \le k < j} t_i : O_{q_i} \to t_j : I_{q_j}$ holds. It is then easy to see that "well-typed programs do not go wrong" as defined in [31]. Namely, an application of one step of SLD-resolution to a well-typed query results always in a new well-typed query. This does not say, however, anything about whether the query succeeds, fails or loops.

The definition above refers to the operational semantics of logic programs based on left to right execution. There is also a more declarative (cf. [32], see also Theorem 1) intuition behind it: Intuitively, the judgments say that if a query has the correct input type and its call terminates successfully, then the computed answer has the correct output type.

Definition 5 (Type checking). *The type-checking problem is to decide for a given program $\mathcal{P}$ and directional type $\mathcal{T}$, whether $\mathcal{P}$ is well-typed wrt. $\mathcal{T}$.*

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 $(list, list, \top) \to (list, list, list)$, where *list* denotes the set of all lists and $\top$ is the set of all terms, but also $(\top, \top, list) \to (list, list, list)$, as well as $(\top, \top, \top) \to (\top, \top, \top)$. (Recall that *append* is seen as a unary predicate here, and $(list, list, list)$ is a set of terms which are triples of lists.) A predicate defined by a single fact $p(X)$ has a directional type $\tau \to \tau$ for all types τ.

Example 2. We show that $(list, list, \top) \to (list, list, list)$ well-types the predicate *append* defined above. For the first clause, $append([], L, L)$, we have to show only one judgment, namely

$$append([], L, L) : (list, list, \top) \to append([], L, L) : (list, list, list).$$

The condition we have to check here is a tautology: the assumption that $[]$ is a member of $list$ implies that $[]$ is a member of $list$, and the assumption that L is a member of both $list$ and $\top$ implies that L is a member of $list$. For the second clause, $append([X|Xs], Y, [X|Z]) \leftarrow append(Xs, Y, Z)$ we have two judgments:

$$append([X|Xs], Y, [X|Z]) : (list, list, \top) \to append(Xs, Y, Z) : (list, list, \top),$$

and

$$\begin{aligned} append([X|Xs], Y, [X|Z]) : (list, list, \top), append(Xs, Y, Z) : (list, list, list) \\ \to append([X|Xs], Y, [X|Z]) : (list, list, list). \end{aligned}$$

The first one follows from the observation that if $[X|Xs]$ is a list then Xs is a list. The second, from the observation that if Z is a list then $[X|Z]$ is a list.

A similar reasoning can be used to show that $((list, list, \top) \cup (\top, \top, list)) \to (list, list, list)$ well-types *append*. Then both $append([a, b, X], [c], L)$ and $append(X, Y, [a, b, c])$ are well-typed queries while $append([a], X, Y)$ is not.

We do not use discriminative types in this paper. We include the definition below to show what the contribution of the paper is. The notion of a path-closed set below originates from [22]. It is equivalent to other notions occurring in the literature: tuple-distributive [30, 35], discriminative [1], or deterministic.

Definition 6 (Discriminative type). *A regular set of ground terms is called* path-closed *if it can be defined by a* deterministic *top-down tree automaton. 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.

A deterministic top-down 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 $list(nil)$.

There are, however, many regular types which are not discriminative. The simplest is the set $\{f(a, a), f(b, b)\}$. Another simple example of a regular but not path-closed set is given in Example 2: it is the set consisting of triples $\langle x, y, z\rangle$ where either x and y are lists and z is any term or x and y are any terms and z is a list (which is useful for typing of the predicate *append* used either for concatenating of the lists x and y or for splitting the list z).

The use of general regular types has also other advantages: it gives us overloading for free. For example, if an operator like $+$ is used in addition of both integers and reals, the corresponding automaton may have simply both transitions $+(\mathsf{int}, \mathsf{int}) \to \mathsf{expr}$ and $+(\mathsf{real}, \mathsf{real}) \to \mathsf{expr}$.

Further motivation for studying regular but not discriminative types comes from program verification. Several papers, including [13, 23, 34] modeled transition systems as logic programs. In many cases safety properties can be tested by type checking: it is enough to prove that some predicates have types of the form $Goodstates \to Goodstates$ where $Goodstates$ is a set which does not contain unsafe states. For example, if we reason about mutual exclusion of two concurrent processes, the set $Goodstates$ could contain three terms: $state(noncritical, noncritical)$, $state(noncritical, critical)$ and $state(critical, noncritical)$. However, any discriminative set containing both terms $state(noncritical, critical)$ and $state(critical, noncritical)$ must also contain the term $state(critical, critical)$ and thus we cannot verify mutual exclusion within such a type system. It is known (cf. [13]) that regular (not limited to discriminative) sets can capture all temporal properties expressible in the logic CTL for all finite systems as well as for some infinite ones, like pushdown or some parameterized systems. Since most model-checkers are limited to finite-state systems, there is a good potential for applications of the logic-programming approach to the infinite case. But to apply a type system for verification we need the full power of regular sets.

3 Directional type checking

In this section we prove that the directional type checking for logic programs wrt. general regular types is DEXPTIME-complete and fixed-parameter linear.

We start with recalling a technique used in [12]. We transform the well-typedness condition in Definition 4 into a logic program $\mathcal{P}_{InOut}$ by replacing $t : I_p$ with the atom $p^{In}(t)$ and $t : O_p$ with $p^{Out}(t)$.

Definition 7 ($\mathcal{P}_{InOut}$, 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 $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ in $\mathcal{P}$, $\mathcal{P}_{InOut}$ contains the n clauses defining in-predicates corresponding to each atom in the body of the clause,*

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

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

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

The program above is known in the literature as a magic-set transformation of the initial program $\mathcal{P}$. It was used (among other things) to obtain more precise information about answers computed by the program if the queries are restricted to some specific form. If we denote by $\mathcal{P}_{In}$ a program that defines some p^{In} predicates (intuitively, the queries to the program $\mathcal{P}$ are then restricted to those defined in the program $\mathcal{P}_{In}$) then it is easy to observe that

$$[\![p^{Out}]\!]_{\mathcal{P}_{In} \cup \mathcal{P}_{InOut}} = [\![p]\!]_{\mathcal{P}} \cap [\![p^{In}]\!]_{\mathcal{P}_{In} \cup \mathcal{P}_{InOut}},$$

which intuitively means that an atom $p^{Out}(t)$ is in the least model of the transformed program if and only if $p(t)$ is in the least model of the initial program and $p^{In}(t)$ is allowed as a query.

The following theorem is proved in [12]. Essentially, it says that a directional type of the form $\mathcal{T} = (Sat(I_p) \rightarrow Sat(O_p))_{p \in \mathsf{Pred}}$, for ground types $I_p, O_p \subseteq T_\Sigma$, satisfies required type judgments if and only if the corresponding directional *ground* type $\mathcal{T}_g = (I_p \rightarrow O_p)_{p \in \mathsf{Pred}}$ does.

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

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

(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 $\mathcal{P}_{InOut}$.

Note that the theorem above connects directional types with arbitrary models of the type program, not only with the least model. Since every clause in this program contains occurrences of predicates p^{In} and there are no facts defining these predicates, the least model is empty, which corresponds to the trivial directional type $\emptyset \rightarrow \emptyset$ (and expresses that a program without input does not produce output). On the other extremity we have the whole Herbrand base, which is also a model of the type program and corresponds to the trivial type $\top \rightarrow \top$.

3.1 Exponential upper bound

Note that a subset of the Herbrand base is a model of a logic program if and only if it is a model of each clause of the program. Thus, as an immediate consequence of Theorem 1 above we obtain that the type-checking problem for directional types reduces to the following model-checking problem.

Problem 1. Given a clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ and a family of regular sets $T_{p_0}, T_{p_1}, \ldots, T_{p_n}$, decide whether the set $\bigcup_{i=0}^{n} \{p_i(t) \mid t \in T_{p_i}\}$ is a model of the clause.

The problem above is closely related to another problem known in the theory of tree automata (in particular, for reasoning about ground reducibility, see [15]), namely, if for a given term t and regular set T there exists a ground instance of t in T. We use similar techniques to prove its decidability. However, since we want to carefully analyze its complexity, we find it easier to present a direct proof rather than to find a suitable reduction. For the decidability proof we need the following lemma.

Lemma 1. *Let $\mathcal{A}_i = \langle \Sigma, Q_i, \Delta_i, F_i \rangle$ for $i = 0, \ldots, n$ be tree automata with disjoint sets of states, and let $\# \notin \Sigma$ be a fresh function symbol of arity $n+1$. There exists a tree automaton $\mathcal{A} = \langle \Sigma \cup \{\#\}, Q, \Delta, F \rangle$ such that*

- *$\mathcal{A}$ is bottom-up deterministic, and*
- *all states of $\mathcal{A}$ are reachable, and*
- *$\mathcal{A}$ recognizes the set $\#(T_\Sigma - \mathcal{L}(\mathcal{A}_0), \mathcal{L}(\mathcal{A}_1), \ldots, \mathcal{L}(\mathcal{A}_n))$, and*
- *$\mathcal{A}$ can be effectively constructed from $\mathcal{A}_0, \ldots, \mathcal{A}_n$ in single exponential time.*

Proof. The idea of the proof below is to use standard complementation and determinisation methods to construct an automaton $\mathcal{A}' = \langle \Sigma \cup \{\#\}, Q', \Delta', F' \rangle$ that satisfies all conditions except reachability of states. The only problem here is that we have to complement and determinize at the same time to avoid a doubly-exponential blowup. Then we obtain $\mathcal{A}$ by removing non-reachable states from $\mathcal{A}'$. The detailed construction is as follows.

We can assume that $\mathcal{A}_0$ is a complete automaton, otherwise we can simply add a new non-final state q (so-called "dead state") to Q_0 and all possible transitions with q on the right-hand side to Δ_0.

Let $Q' = 2^{Q_0 \cup \ldots \cup Q_n} \cup \{s_{\text{fin}}\}$ be the powerset of $Q_0 \cup \ldots \cup Q_n$ plus one additional state s_{fin}, which is the only final state of $\mathcal{A}'$, that is $F' = \{s_{\text{fin}}\}$. For $s_1, \ldots, s_k \in Q'$ and k-ary $f \in \Sigma$ we define that $f(s_1, \ldots, s_k) \to s \in \Delta'$ if s is the set

$$\{q \in Q_0 \cup \ldots \cup Q_n \mid \exists q_1 \in s_1 \ldots \exists q_k \in s_k \; f(q_1, \ldots, q_k) \to q \in \Delta_0 \cup \ldots \cup \Delta_n\}.$$

For $s_0, \ldots, s_n \in Q'$ we define that $\#(s_0, \ldots, s_n) \to s_{\text{fin}} \in \Delta'$ if

$$s_0 \cap F_0 = \emptyset, s_1 \cap F_1 \neq \emptyset, \ldots, s_n \cap F_n \neq \emptyset.$$

Finally we define Q as the set of reachable states from Q' (it is well-known that reachability for tree automata can be tested in polynomial time), Δ as the restriction of Δ' to Q, and F as F'.

The correctness of the construction follows immediately from the observation that for $i = 0, \ldots, n$, the automaton

$$\mathcal{A}'_i = \langle \Sigma \cup \{\#\}, Q, \Delta, \{s \in Q \mid s \cap F_i \neq \emptyset\} \rangle$$

recognizes exactly the set $\mathcal{L}(\mathcal{A}_i)$, and $\mathcal{A}'_0$ restricted to Σ is complete. □

Decidability of Problem 1. Let the clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ and the family of regular sets $T_{p_0}, T_{p_1}, \ldots, T_{p_n}$ be an instance of Problem 1. We did not specify here the formalism in which the sets $T_{p_0}, \ldots, T_{p_n}$ are given, but without loss of generality we can assume that the automata recognizing them are known. The translation from other formalisms like ground set expressions from [1] or regular grammars from [16] is straightforward.

The idea of the proof is to test the emptiness of the intersection of the automaton constructed in Lemma 1 with the set of instances of the term $\#(t_0, \ldots, t_n)$. Due to non-linear occurrences of variables in $\#(t_0, \ldots, t_n)$ this last set is, however, not regular. For our purposes it is enough, however, if we assign the same state of an automaton to each occurrence of the same variable.

Lemma 2. *Let $\mathcal{A} = \langle \Sigma, Q, \Delta, F\rangle$ be a deterministic bottom-up tree automaton without unreachable states, recognizing $\#(T_{\Sigma-\{\#\}} - T_0, T_1, \ldots, T_n)$, as constructed in Lemma 1. Then the set $\bigcup_{i=0}^{n}\{p_i(t) \mid t \in T_{p_i}\}$ is not a model of the clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ if and only if there exists a mapping $\theta : \mathsf{Var}(\#(t_0, \ldots, t_n)) \to Q$ such that the term $\#(t_0, \ldots, t_n)\theta$ is accepted by the automaton $\mathcal{A}$.*

Proof. The above set is not a model of the clause if and only if there exists a substitution $\sigma : \mathsf{Var}(\#(t_0, \ldots, t_n)) \to T_{\Sigma-\{\#\}}$ such that $t_1\sigma \in T_{p_1}, \ldots, t_n\sigma \in T_{p_n}$ and $t_0\sigma \notin T_{p_0}$. This is equivalent to the existence of such a σ that the automaton $\mathcal{A}$ accepts the term $\#(t_0, \ldots, t_n)\sigma$. Thus it is enough to prove the equivalence of the last condition with the acceptance of $\#(t_0, \ldots, t_n)\theta$ by $\mathcal{A}$.

Now we prove this equivalence. Suppose that $\mathcal{A}$ accepts $\#(t_0, \ldots, t_n)\sigma$ with a run ρ. Note that by the determinism of $\mathcal{A}$, there is only one possible run of $\mathcal{A}$ on $\#(t_0, \ldots, t_n)\sigma$, and for each occurrence of $x\sigma$ the state assigned by ρ is the same, and thus we can speak about states assigned to terms (as opposed to occurrences of terms). Taking $\theta(x) = \rho(\sigma(x))$ we obtain $\rho(\#(t_0, \ldots, t_n)\theta) = \rho(\#(t_0, \ldots, t_n)\sigma) \in F$ and the automaton accepts $\#(t_0, \ldots, t_n)\theta$.

Conversely, suppose there exists θ such that $\#(t_0, \ldots, t_n)\theta$ is accepted by $\mathcal{A}$. Since all states in Q are reachable, there exists a tree t_x accepted by the state $\theta(x)$. Putting $\sigma(x) = t_x$ for all $x \in \mathsf{Var}(\#(t_0, \ldots, t_n))$ we obtain a σ such that $\mathcal{A}$ accepts the term $\#(t_0, \ldots, t_n)\sigma$. □

Theorem 2. *Problem 1 is decidable in DEXPTIME.*

Proof. This is a consequence of the Lemma 2 above: there are $|Q|^{|\mathsf{Var}(\#(t_0,\ldots,t_n))|}$ possible mappings θ; this number is exponential in the size of the input, since $|Q|$ is exponential and $|\mathsf{Var}(\#(t_0, \ldots, t_n))|$ is linear; each such θ can be tested in polynomial time. □

The following corollary is a direct consequence of Theorems 1 and 2, and the DEXPTIME-hardness result from [12].

Corollary 1. *Directional type checking for logic programs wrt. arbitrary regular types is DEXPTIME-complete.*

3.2 Parameterized complexity of type checking

Let us recall the DEXPTIME-hardness proof for the type-checking of logic programs wrt. discriminative types. It is based on a reduction from the emptiness problem for intersection of deterministic top-down tree automata [37]. It is shown that the program consisting of a single clause $p(X, \ldots, X)$ is well-typed wrt. $(T_1, \ldots, T_n) \to \emptyset$ if and only if the intersection $T_1 \cap \ldots \cap T_n$ is empty. For the hardness proof the sets $T_1, \ldots, T_n$ are chosen as discriminative regular sets of trees, whose intersection encodes computation of an alternating Turing machine with polynomialy bounded tape.

What strikes in this construction is that the program used here is very short (it is only one fact) while the types are very large (the encoding of the Turing-machine computation is done in the types, not in the program). The situation in practice looks usually quite opposite: the type-checking problems are usually applied to large programs and rather small types. A natural way to approach such problem is to study its parameterized complexity [18].

A parameterized problem takes as input a pair $\langle x, k\rangle$ where x is a word (in our case the encoding of a logic program and a directional type) and k is a positive integer. Such a problem is called fixed-parameter linear if there exists a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that decides the problem and runs in time $f(k)|x|$.

In the formulation of the problems below, $|\mathcal{T}|$ denotes the size of $\mathcal{T}$. Formally, it is the sum of the lengths of the encodings of the automata recognizing the regular sets occurring in $\mathcal{T}$. Similarly, $|T|$ denotes the size of T (the length of the encoding of the automaton recognizing T).

Problem 2 (Parameterized type-checking).
Instance: a logic program $\mathcal{P}$ and a directional type $\mathcal{T}$
Parameter: $|\mathcal{T}|$
Question: is $\mathcal{P}$ well-typed wrt. $\mathcal{T}$?

Theorem 3. *The parametrized type-checking problem is decidable in time $O(c^{|\mathcal{T}|} \cdot |\mathcal{P}|)$ for some constant c that does not depend on $\mathcal{P}$.*

The proof of this theorem follows directly from Lemma 3

Problem 3 (Parametrized version of Problem 1).
Instance: a clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ and a family of regular sets $T_{p_0}, T_{p_1}, \ldots, T_{p_n}$
Parameter: $\sum_{p_i \in \{p_1,\ldots,p_n\}} |T_{p_i}|$
Question: is the set $\bigcup_{i=0}^{n} \{p_i(t) \mid t \in T_{p_i}\}$ a model of the clause?

Lemma 3. *Problem 3 is decidable in time $O(c^k m)$ for some constant c that does not depend on m, where k is the parameter and m is the size of the clause.*

Proof. The idea is again to use Lemma 2. We use notations from Lemmas 1 and 2. We traverse the term $\#(t_0, \ldots, t_n)$ top-down checking which assumptions have to be made on the value of the run on subterms to make the term accepted by the automaton. These checks will succeed if the assumptions about the run on variables give raise to a function from variables to states of the automaton.

Consider a set of pairs of the form $\{\langle q_1, s_1\rangle, \ldots, \langle q_k, s_k\rangle\}$ where q_i is a state of the automaton $\mathcal{A}$ and s_i is a subterm of $\#(t_0, \ldots, t_n)$. Intuitively, this set will express the information "if there exists a run ρ of the automaton $\mathcal{A}$ such that $\rho(s_i) = q_i$ then $\mathcal{A}$ accepts $\#(t_0, \ldots, t_n)$". We call such a set S *flat* if all terms occurring in S are variables. A flat set S is *inconsistent* if it contains two different pairs $\langle q, x\rangle$ and $\langle q', x\rangle$ with the same variable x and different states q, q'; otherwise it is consistent. A flat and consistent set defines a function assigning states to variables.

Consider a function check assigning a boolean value to such sets of pairs, defined recursively as follows.

$$\mathsf{check}(S) = \begin{cases} \mathit{true}, & \text{if } S \text{ flat and consistent} \\ \mathit{false}, & \text{if } S \text{ flat and inconsistent} \\ \displaystyle\bigvee_{f(q_1,\ldots,q_k)\to q\in\Delta} \mathsf{check}(S - \{\langle q, f(s_1,\ldots,s_k)\rangle\} \cup \{\langle q_1, s_1\rangle, \ldots, \langle q_k, s_k\rangle\}), & \\ & \text{otherwise} \end{cases}$$

We claim that

1. $\mathsf{check}(\{\langle s_{\mathrm{fin}}, \#(t_0, \ldots, t_n)\rangle\}) = \mathit{true}$ if and only if there exists a function $\theta : \mathsf{Var}(\#(t_0, \ldots, t_n)) \to Q$ such that the term $\#(t_0, \ldots, t_n)\theta$ is accepted by the automaton $\mathcal{A}$.
2. the value of $\mathsf{check}(\{\langle s_{\mathrm{fin}}, \#(t_0, \ldots, t_n)\rangle\})$ can be computed in time $O(|\#(t_0, \ldots, t_n)| \cdot |\mathcal{A}|)$

The first part can be easily proved by induction on the structure of the term $\#(t_0, \ldots, t_n)$. The run of the automaton must assign the final state s_{fin} to the term $\#(t_0, \ldots, t_n)$, which is expressed by the pair $\langle s_{\mathrm{fin}}, \#(t_0, \ldots, t_n)\rangle$; each computation step of the automaton must agree with some transition in Δ, which is expressed by the disjunction over matching transitions in the definition of check; finally the condition that θ is a function is expressed by the consistency of the set S. Note also that by the associativity and commutativity of disjunction, the value of check does not depend on the choice of the non-flat element $\langle q, f(s_1, \ldots, s_n)\rangle$ from S.

For the second part, note that for each subterm s of $\#(t_0, \ldots, t_n)$ there are at most $|\Delta|$ calls to the function check that correspond to decomposing of the term s. Since there are exactly $|\#(t_0, \ldots, t_n)|$ such subterms, the whole work is done in time $O(|\#(t_0, \ldots, t_n)| \cdot |\mathcal{A}|)$. □

3.3 Incrementality and infinite signature

A literal application of the algorithm presented above might lead to the following problem. Suppose that some program is well-typed and we increment it by adding a new, completely independent, fragment defining a new predicate. The new fragment may contain new function symbols, which did not occur in the original program. Since a signature is a part of the definition of a tree automaton, the old type-check was done with automata over smaller signature, and one could argue that now the type-checking procedure has to be rerun from scratch.

However, it is fairly straightforward to extend tree automata to deal with infinite signature. We can simply consider an infinite signature Σ_{inf} containing Σ, add a new state q_{any} to the automaton and say that the transition relation Δ implicitly contains all transitions of the form $f(\ldots) \to q_{\mathrm{any}}$ for all $f \in \Sigma_{inf}$. Such an automaton has still finite set of states and infinite (but finitely representable) transition relation.

With such an extension of tree automata our algorithm is still correct (a little bit of work has to be done to correctly reason about implicit transitions and reachable states during the determinization step); it is still fixed-parameter linear (when traversing the term $\#(t_0, \ldots, t_n)$ there is no need to look at function symbols that do not occur in this term).

Another problem of the same nature is that numbers (integers or reals) do not form a regular set. In order to extend tree automata to deal with these sets it is enough to treat each number as a constant symbol, add two states int and real and infinitely many implicit transitions $i \to \mathsf{int}$ and $r \to \mathsf{real}$ for all integers i and reals r.

4 Conclusion

We proved the decidability in DEXPTIME and fixed-parameter linearity of directional-type checking for logic programs wrt. general regular types. This solves a problem that was open since 1994 and improves several earlier partial solutions.

The procedure we presented is optimal from the complexity point of view, it is also incremental. This, together with linear complexity in the size of program gives us a hope that the type system may be usable in practice.

There are some obvious directions for the future work. One is the implementation of the system to see how it behaves in practice. Further, an extension to constraint logic programming, negation etc. would be interesting. The extension to polymorphic types seems not to be very difficult.

Acknowledgments. I thank Andreas Podelski for interesting discussions and the anonymous referees for their comments on the paper.

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. A. Caron, J. Coquidé, and M. Dauchet. Encompassment properties and automata with constraints. In C. Kirchner, editor, *5th international conference on Rewriting Techniques and Applications*, LNCS 690, pages 328–342, Montréal, 1993.
10. W. Charatonik. Automata on DAG representations of finite trees. Technical Report MPI-I-1999-2-001, Max-Planck-Institut für Informatik, Mar. 1999. www.mpi-sb.mpg.de/~witold/papers/dag.ps.
11. W. Charatonik, F. Jacquemard, and A. Podelski. Directional type checking for logic programs is decidable, 1998. Unpublished note.
12. W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the Fifth International Static Analysis Symposium (SAS)*, LNCS 1503, pages 278–294, Pisa, Italy, 1998. Springer-Verlag.
13. W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 358–375, Lisbon, Portugal, March-April 1998. Springer-Verlag.
14. W. Charatonik, A. Podelski, and J.-M. Talbot. Paths vs. trees in set-based program analysis. In *27th Annual ACM Symposium on Principles of Programming Languages*, Jan. 2000.
15. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. www.grappa.univ-lille3.fr/tata.
16. P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–189. MIT Press, 1992.
17. 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.
18. R. G. Downey and M. Fellows. *Parameterized complexity.* Monographs in computer science. Springer, New York, 1999.
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. G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: Temporal versus deductive reasoning in verification. Technical Report DBAI-TR-98-22, Institut für Informationssysteme, Technische Universität Wien, December 1998.
24. N. Heintze. *Set based program analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.
25. 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.
26. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–156. MIT Press, 1992.
27. F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
28. P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, Mar. 1995.
29. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, July 1992.
30. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
31. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
32. L. Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, September 1996.
33. F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.
34. Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification (CAV'97)*, LNCS 1254. Springer-Verlag, June 1997.
35. 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.
36. P. Rychlikowski and T. Truderung. Polymorphic directional types for logic programming. http://www.tcs.uni.wroc.pl/~tomek/dirtypes/, 2000.
37. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52:57–60, 1994.
38. W. Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter Automata on Infinite Objects, pages 134–191. Elsevier, 1990.
39. E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10:125–153, 1991.