INFORMATIK

Automata on DAG Representations of Finite Trees

Witold Charatonik

MPI–I–99–2–001 March 1999

FORSCHUNGSBERICHT RESEARCH REPORT

MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Im Stadtwald 66123 Saarbrücken Germany

Author's Address

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
Germany
E-Mail: `witold@mpi-sb.mpg.de`
URL: `www.mpi-sb.mpg.de/~witold`
Phone: +49 681 9325 204

Acknowledgements

I thank Andreas Podelski and Jean-Marc Talbot for their helpful comments and discussions.

Abstract

We introduce a new class of finite automata. They are usual bottom-up tree automata that run on DAG representations of finite trees. We prove that the emptiness problem for this class of automata is NP-complete. Using these automata we prove the decidability of directional type checking for logic programs, and thus we improve earlier results by Aiken and Lakshman. We also show an application of these automata in solving systems of set constraints, which gives a new view on the satisfiability problem for set constraints with negative constraints.

Keywords

Tree Automata, Directed Acyclic Graphs, Types in Logic Programming, Semantics of Logic Programs, Set Constraints

1 Introduction

We introduce a new class of finite automata, which we call automata on t-dags. They are usual bottom-up tree automata that run on DAG representations of ground terms over given signature. The class of languages recognizable by these automata contains all DAG representations of regular sets of terms and is closed under union and intersection. We prove decidability of the membership and emptiness problems for these automata. The emptiness test presented here, based on ideas from [15], is an adaptation of the standard pumping techniques to the case of t-dags. In Sections 2 and 3 we introduce our automata, show their basic properties and prove the NP-completeness of the emptiness problem.

The main difference between the introduced class and the automata on directed acyclic graphs that are known in the literature [28, 29, 30, 12, 33, 35, 36] is that the considered DAGs have to maximally share structure, i.e., they do not contain two isomorphic copies of the same subgraph. Moreover, graphs from [12, 28, 29] are always planar (we do not require planarity), and graphs from [30] are infinite. Tiling systems from [35, 36] work on much more general classes of graphs, in particular the emptiness problem for them is undecidable.

Our automata are closely related to tree set automata of [24, 25], which accept mappings from Herbrand universe. The requirement that the accepted object is a function has the same consequence as the sharing of structure in case of DAGs: the automaton cannot assign two different states to two occurrences of the same [representation of a] tree. The difference is that both kinds of automata recognize different kinds of objects; in particular, we do not see any connection between tree set automata and directional types in logic programs. However, our emptiness test gives an alternative (and as we believe, a simpler) proof of the decidability of the emptiness problem for tree set automata.

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 [34], as mode dependencies in [14], and simply as types in [9, 7, 8]. Our use of the terminology "directional type" stems from [5]. More pointers to the literature can be found in [13].

It is pointed in [5] that the type checking problem is undecidable for arbitrary types. Therefore one has to restrict himself to regular (or even more restricted, e.g. discriminative) types. In [5], Aiken and Lakshman present an algorithm for automatic type checking of logic programs wrt. given (regular) directional types. The algorithm runs in 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 regular types). Another decidability proof (without complexity analysis) for type-checking for discriminative directional types is given in [13]. In [19] we proved that directional type checking wrt. discriminative types is DEXPTIME-complete and gave an algorithm for *inferring* (regular, not necessarily discriminative) directional types.

The methods used in the mentioned papers are not strong enough to prove the decidability of directional type checking wrt. general regular types. In Section 4, using automata on t-dags, we prove the decidability of this problem. This improves the results by Aiken and Lakshman [5], Boye [13], and Charatonik and Podelski [19], where decidability is restricted to discriminative types.

Set constraints. There are several kinds of automata used in solving systems of set constraints: tree set automata with free variables [24], tree set automata [25], Σ-graph automata [26], as well as standard tree automata [17, 18, 22]. These non-standard ones are seen as acceptors for mappings from the Herbrand universe over given signature to some finite set. Usually it is quite difficult to understand the essence of the difference between them and standard automata. In particular, the definition of Σ-graph automaton is rather sophisticated. In this paper, we extract the essence of these three kinds of automata: we remove all non-standard constructs and replace them by a single requirement that an automaton runs on DAG representations of trees. This gives a very simple and intuitive definition of the automaton.

The automata used for solving negative set constraints have rather complicated procedure for testing emptiness, with a very sophisticated correctness proof. The proof presented here, based on ideas from [15], is an adaptation of the standard pumping techniques to the case of t-dags.

In Section 5 we present an application of t-dag automata to solving system of set constraints. It gives a unified view for three different methods for solving positive set constraints and simplifies rather complicated proofs of decidability of the problem for positive and negative set constraints [25, 4, 15]. The key point of the reduction is the decidability of the emptiness test for t-dag automata.

2 Preliminaries

A subgraph G' of a directed graph G is *closed* if G' contains with every node v all the successors of v in G.

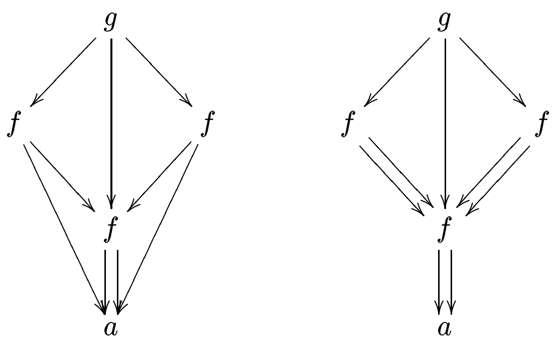

Figure 1: A t-dag representing the term $g(f(a, f(a, a)), f(a, a), f(f(a, a), a))$ and a graph not being a t-dag.

Definition 1 (t-dag) *A DAG representation of a ground term (t-dag in short) over a signature Σ is a directed acyclic ordered graph whose nodes are labeled with function symbols from Σ such that*

- *if a node is labeled with a function symbol of arity n then it has n immediate successors in the graph, and*
- *it does not contain two different isomorphic closed subgraphs.*

DAG representations of trees are known in the literature on unification. The second condition in the definition above is known as *maximal sharing of structure.* The assumption that a t-dag is ordered (that is, the successors of each node are ordered) is needed to assure that the t-dags representing $f(a, b)$ and $f(b, a)$ are not isomorphic. Figure 1 shows an example of a t-dag, and a graph which is not a t-dag because it contains two isomorphic copies of the graph representing $f(f(a, a), f(a, a))$.

Definition 2 (t-dag automaton) *A t-dag automaton is a tuple $\langle \Sigma, Q, F, \Delta \rangle$ where Σ is a finite signature, Q is a finite set of states, $F \subseteq Q$ is the set of final states, and Δ is a set of transitions of the form $f(q_1, \ldots, q_n) \to q$ with $q, q_1, \ldots, q_n \in Q$, $f \in \Sigma$ and n being the arity of f.*

An automaton is called *complete* if for each $f \in \Sigma$ and each sequence $q_1, \ldots, q_n$ of the appropriate length (the arity of f) there is $q \in Q$ such that $f(q_1, \ldots, q_n) \to q$ belongs to Δ. It is called *deterministic* if for each f and $q_1, \ldots, q_n$ there exists at most one such q, and *nondeterministic* otherwise.

Note that up till now there is no difference between t-dag automata and standard bottom-up tree automata. The difference is that t-dag automata run on t-dags and not trees.

Definition 3 (run) *A run of a t-dag automaton* $\langle \Sigma, Q, F, \Delta \rangle$ *on a given t-dag* G *is a mapping* r *from the set of nodes of* G *to the set of states* Q *such that for each node* v *and each* $f \in \Sigma$*, if* v *is labeled with* f *and* $v_1, \ldots, v_n$ *are immediate successors of* v*, then* Δ *contains a transition* $f(r(v_1), \ldots, r(v_n)) \to r(v)$*. A run is successful if it maps the root of* G *to a final state.*

We say that a state q *accepts* a sub-t-dag rooted at a node v for a given run r if $r(v) = q$. An automaton $\mathcal{A}$ accepts a t-dag G if there exists a successful run of $\mathcal{A}$ on G.

Example 4 *The automaton* $\langle \{a, f(\cdot,\cdot)\}, \{q_1, q_2, q\}, \{q\}, \{a \to q_1,\ a \to q_2, f(q_1, q_2) \to q\} \rangle$ *accepts* $f(a, a)$ *as a tree automaton, but as a t-dag automaton it does not accept any t-dag representation of any tree.*

2.1 Recognizable sets of t-dags

A set T of t-dags is recognizable if there exists a t-dag automaton $\mathcal{A}$ such that $T = \{G \mid \mathcal{A} \text{ accepts } G\}$. We write then $T = \mathcal{L}(\mathcal{A})$

Proposition 5 *For every regular set* T *of trees, the set* t-dag(T) *of t-dag representations of trees from* T *is recognizable.*

Proof. T can be recognized by a deterministic tree automaton $\mathcal{A}$. As a t-dag automaton, $\mathcal{A}$ recognizes t-dag(T). □

Proposition 6 *The class of recognizable sets of t-dags is closed under union and intersection.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, F, \Delta \rangle$ and $\mathcal{A}' = \langle \Sigma', Q', F', \Delta' \rangle$ be the automata recognizing the sets T and T', respectively. We assume that $Q \cap Q' = \emptyset$.

The set $T \cup T'$ is recognized by the automaton

$$\mathcal{A}_\cup = \langle \Sigma \cup \Sigma', Q \cup Q', F \cup F', \Delta \cup \Delta' \rangle.$$

The set $T \cap T'$ is recognized by the automaton

$$\mathcal{A}_\cap = \langle \Sigma \cap \Sigma', Q \times Q', F_\cap, \Delta_\cap \rangle,$$

where $F_\cap = \{\langle q, q' \rangle \mid q \in F, q' \in F'\}$ and

$$\begin{aligned}\Delta_\cap = \{ f(\langle q_1, q'_1 \rangle, \ldots, \langle q_n, q'_n \rangle) \to \langle q, q' \rangle \mid \\ f(q_1, \ldots, q_n) \to q \in \Delta, \quad f(q'_1, \ldots, q'_n) \to q' \in \Delta' \}.\end{aligned}$$

□

As we have seen above, the standard methods from tree automaton theory show that the class of recognizable sets of t-dags is closed under union and intersection. These methods fail however to show closedness under complementation. In case of trees, one proves recognizability of the complement of a regular set by determinisation of the corresponding automaton. In case of the automaton from Example 4 the determinisation procedure yields the automaton $\langle\{a, f(\cdot,\cdot)\}, \{\{q_1, q_2\}, \{q\}\}, \{\{q\}\}, \{a \to \{q_1, q_2\},\ f(\{q_1, q_2\}, \{q_1, q_2\}) \to \{q\}\}\rangle$, which is correct for tree automata, but not for t-dag automata. We do not know answers to the following questions.

Questions.

1. *Are t-dag automata determinisable?*

2. *Is the class of recognizable sets of t-dags closed under complementation?*

3. *Does there exist a recognizable set T of t-dags such that the set tree(T) of trees represented in T is not regular?*

A positive answer to the first question would imply a positive answer to the second one and negative answer to the third one. A negative answer to the first question is known for the related classes of Σ-graph automata [26] and general automata on directed acyclic graphs [33]. The counter-examples presented in both these papers do not work here. In case of [26], the example exploits the infinite character of accepted objects (and a particular accepting condition). There are two counter-examples in [33]. One of them essentially depends on isomorphic subgraphs, and the other one uses the fact that the input is restricted to graphs representing $(n \times n)$-grids. These grids can be seen as t-dags, but the set of $(n \times n)$-grids is not recognizable, which can be proved by use of the pumping lemma (Claim 3 in Section 3).

Proposition 7 *The membership problem for t-dag automata is decidable in nondeterministic linear time.*

Proof. For a given t-dag G and automaton $\mathcal{A}$ it is enough to guess a successful run of $\mathcal{A}$ on G. $\square$

Note that for tree automata the membership problem is decidable in polynomial time. The proof uses however a determinisation technique (one computes on the fly a run of the deterministic version of the given automaton on a given tree) and therefore does not work for t-dag automata.

3 Emptiness problem

In this section we prove NP-completeness of the emptiness problem for t-dag automata, that is, the problem of answering the question whether there exists a t-dag accepted by a given automaton. For the upper bound, we use here a sort of pumping lemma technique; the main idea comes from [15]. We present it here as natural extension of the analogous proof for tree automata. The lower bound for the emptiness problem is rather easy encoding of the SAT problem.

Theorem 8 *The emptiness problem for t-dag automata is NP-complete.*

The upper bound follows from Theorem 14: it is enough to guess a t-dag of size at most $2|Q|^3$ where Q is the set of states, and a successful run of the automaton on this t-dag. The lower bound is proved in Theorem 16.

3.1 Intuition: tree automata

To make the understanding of our approach easier, we first show our view on the emptiness problem for bottom-up tree automata. We show that if the language recognized by an automaton $\mathcal{A} = \langle \Sigma, Q, F, \Delta \rangle$ is nonempty, then there is a tree of depth at most $|Q|$, accepted by $\mathcal{A}$.

Consider a tree t accepted by $\mathcal{A}$, and a path of maximal length in this tree. If this path is longer then $|Q|$, there must be a state in $|Q|$ assigned to two different nodes v and v' in t, and one can remove all nodes between v and v'. One should not, however, forget here about the paths to other states that are needed to reach the final state and do not lie on the chosen path. The formalization of this method leads to the following notion of a skeleton of a run of an automaton on a tree. For the sake of simplicity we identify here a tree with its graph representation, and a node in this graph with a tree rooted at this node.

Definition 9 *A skeleton of a run r of a tree automaton $\mathcal{A}$ on a tree t is a subgraph G of t such that*

- *G contains the root of t, and*
- *for each node v in G, the maximal subterm of v in G is in G', and*
- *each node v in G is labeled with the transition used by $\mathcal{A}$ to reach v and the position of the maximal subterm of v, and*
- *if a state q is used in a transition labeling a node v then it is provided by a transition labeling some other node below v.*

If any path in a skeleton contains twice the same state, we can remove the appropriate part of the path, just as one does it in pumping lemma for string automata. It is easy to see how one can reconstruct from a skeleton a tree accepted by the automaton.

Example 10 *Consider an automaton with* $\Sigma = \{a, b, g(\cdot), f(\cdot,\cdot)\}$, $Q = \{q_0, q_1, q_2, q\}$, $F = \{q\}$, *and*

$$\Delta = \{a \to q_0,\ \ b \to q_0,\ \ g(q_0) \to q_0,\ \ g(q_1) \to q_1,\\ f(q_0,q_0) \to q_1,\ \ f(q_0,q_0) \to q_2,\ \ f(q_1,q_2) \to q\}.$$

Figure 2 shows an example of a tree accepted by $\mathcal{A}$ *and a skeleton of a run of* $\mathcal{A}$ *on this tree. Figure 3 shows the "pumped-out" skeleton and the accepted tree induced by this skeleton.*

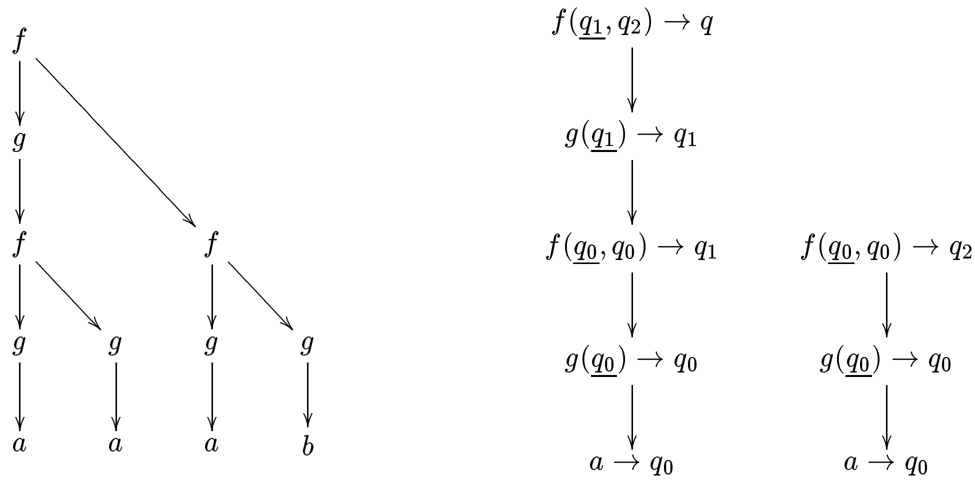

Figure 2: An accepted tree and a skeleton. Underlined states indicate the position of the maximal subterm.

This method does not work for t-dag automata. The reason is that the skeleton may contain two nodes with the same left-hand sides of the labeling transitions, the same paths below them, but different right-hand sides of

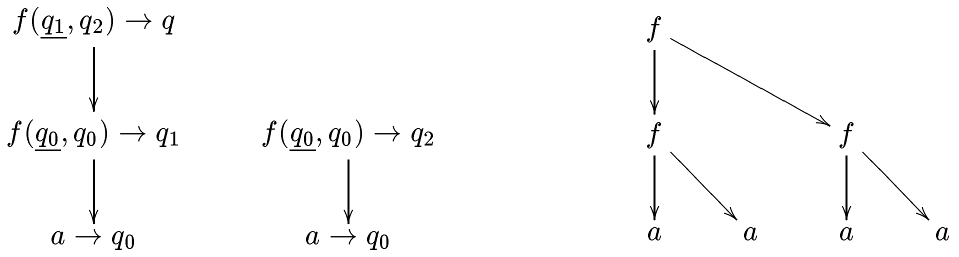

Figure 3: An pumped-out skeleton and a smaller accepted tree.

the transitions (like the nodes labeled $f(\underline{q_0}, q_0) \to q_1$ and $f(\underline{q_0}, q_0) \to q_2$ on Figures 2 and 3). Such two nodes induce the same tree and thus must be identified in a DAG representation. On the other hand, they must be different in order to provide two different states in the run. To overcome this problem we extend the skeleton in order to obtain different graphs that are induced in such nodes. In case of trees, in both nodes labeled with a transitions starting with $f(\underline{q_0}, q_0)$ we used the least tree providing the state q_0. In case of t-dags, we will use two different t-dags (the least and the least but one providing q_0) which we note $q_0[1]$ and $q_0[2]$. The technical problem here is to make sure that we have enough nodes providing given state and to estimate the size of the skeleton.

3.2 Notations

A *leaf* in a t-dag is a node without immediate successors, labeled with a constant symbol. The *main path* for a given node v in G is the longest path leading from v to a leaf in G; if there are several paths of the same (biggest) length, then the leftmost of them is the main one. The length of the main path for a given node is called the *depth* of the node. The depth of a t-dag is the depth of its root.

The *main successor* of a node v is the immediate successor of v lying on the main path for v. The position of the main successor (that is, the number identifying this node in the ordered sequence of successors of v) is called the main position. Note the "leftmost" requirement above; it implies that if $v_1, \dots, v_n$ are the immediate successors of v in G, and v_i lies on the main path for v, then the main paths for $v_1, \dots, v_{i-1}$ are strictly shorter than the path for v_i.

We say that a node v *lies below* a node v' if the main path for v is shorter than the main path for v' (that is, the depth of v is smaller than the depth of v'), and we fix some linear order $\prec$ on nodes that extends the "lies below" partial order.

States occurring on the left-hand side of the arrow in a transition are called *used* by this transition; the state on the right-hand side is called *provided* by this transition. We say that a state q *accepts* a sub-t-dag rooted at a node v (equivalently, that the node v *provides* the state q) for a given run r if $r(v) = q$.

A *pointer* is a pair (q, i), where q is a state and i is a number. We write $q[i]$ instead of (q, i). For a given t-dag G and a run r we say that the pointer $q[i]$ points to the i-th (according to the order $\prec$) node providing the state q. By *index* of a given node v (in symbols, $\mathsf{ind}_r(v)$) we mean such a number i that the pointer $r(v)[i]$ points to v. A *transition with pointers* is an expression of the form $f(q_1[i_1], \dots, q_{k-1}[i_{k-1}], \underline{q_k}, q_{k+1}[i_{k+1}], \dots, q_n) \to q$. We say that such a transition is compatible with the transition $f(q_1, \dots, q_n) \to q$ and the k-th position.

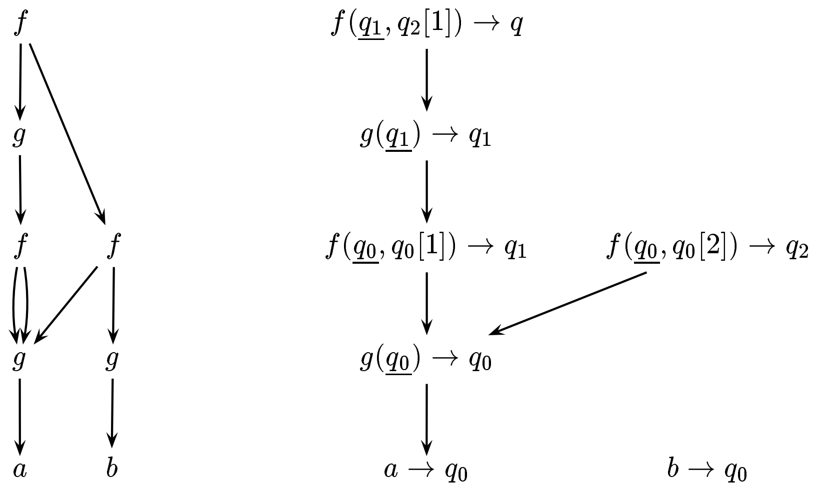

Figure 4: A t-dag and a skeleton

Definition 11 (skeleton) *A skeleton of a run r of an automaton $\mathcal{A}$ on a t-dag G is a subgraph G' of G such that*

- *G' contains the root of G, and*
- *for each node v in G', the main successor of v in G is in G', and*
- *each node v in G' is labeled with a transition with pointers, compatible with the transition used by $\mathcal{A}$ to reach v and the position of the main successor of v, and*
- *if a pointer $q[i]$ is used in a transition labeling a node v then $q[i]$ points to some node v' in G, v' lies below v, and v' is in G', and*
- *the graph induced from G' (see Definition 13 below) is a t-dag.*

Example 12 *Figure 4 shows an example of a t-dag accepted by the automaton from Example 10 (this time we view it as a t-dag automaton) and a skeleton of a successful run r on this t-dag. If v_a is the node labeled with a and v_b is the node labeled with b, and assuming that $v_a \prec v_b$, we have $\mathsf{ind}_r(v_a) = 1, \mathsf{ind}_r(v_b) = 2$, the index of the root is 1 and the indexes of the nodes corresponding to $g(a)$ and $g(b)$ are 3 and 4.*
Note that a skeleton need to be a connected graph.

Definition 13 (induced graph) *A graph induced from a skeleton S is a directed acyclic graph G such that*

- *the set of nodes in G is the same as the set of nodes in S, and*

- *if a node v is labeled with $f(q_1[i_1], \ldots, \underline{q_k}, \ldots, q_n[i_n]) \to q$ in S then*
  - *v is labeled with f in G, and*
  - *the k-th successor of v in G is the successor of v in S, and*
  - *for $j = 1, \ldots, k-1, k+1, \ldots, n$, the j-th successor of v in G is the i_j-th node (according to the ordering $\prec$) providing the state q_j.*

Figure 5 shows an example of a graph induced from a skeleton.

Note that a node in a skeleton cannot have two immediate predecessors with the same left-hand side $f(q_1[i_1], \ldots, \underline{q_k}, \ldots, q_n[i_n])$ of the labels (otherwise the induced graph is not a t-dag).

3.3 Upper bound

Let us fix an automaton $\mathcal{A} = \langle \Sigma, Q, F, \Delta \rangle$, a t-dag G and a successful run r of $\mathcal{A}$ on G. Below we show that if G is big enough then we can find a smaller t-dag G' and a successful run r' on G'.

Theorem 14 *If an automaton $\mathcal{A} = \langle \Sigma, Q, F, \Delta \rangle$ accepts a t-dag then there exists another t-dag with at most $2|Q|^3$ nodes, accepted by $\mathcal{A}$.*

Let r be a successful run of $\mathcal{A}$ on a t-dag G. The idea of the proof is quite simple: first we prove that there exists a skeleton of r. If this skeleton is too big, then we "pump it out". We obtain the other t-dag as the graph induced from the pumped-out skeleton. Figures 4 and 5 give an example of this procedure. Figure 4 shows an example of a t-dag accepted by the automaton from Example 10 (this time as an automaton over t-dags, not trees) and a skeleton of a successful run r on this t-dag. Figure 5 shows the same skeleton after pumping it out, and the induced t-dag, accepted by the same automaton.

Construction of the skeleton S. We define S as the smallest graph satisfying the conditions

- S contains the root of G
- for each node v in S, the main successor of v is in S
- S satisfies the fork condition below
- for each pointer $q[i]$ used in a label of a node in S, the node pointed by $q[i]$ is in S

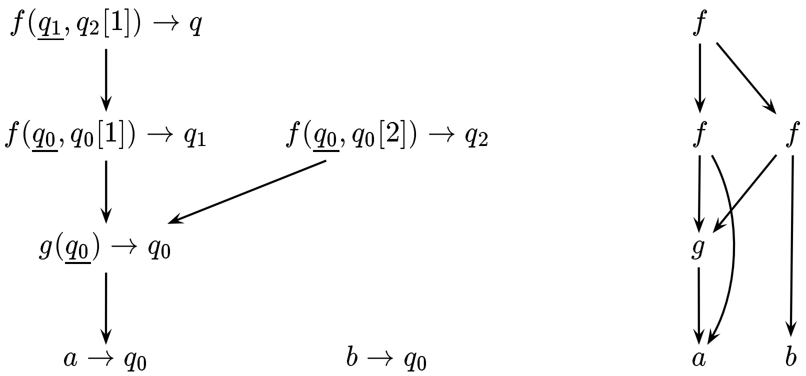

Figure 5: A pumped-out skeleton and the induced t-dag

A *fork* of degree $m \geq 1$ in a skeleton S is a node having m different immediate predecessors with the same left-hand side of the corresponding transition of the automaton (that is, the transition compatible with the label of the node) and the same position of the main successor. On Figure 4 there is one fork of degree 2.

Definition 15 (fork, fork condition) *We say that v is a fork of degree $m \geq 1$ in a skeleton S of a run r on a t-dag G if v has m predecessors $v_1, \ldots, v_m$ with the following properties*

- *for each $i = 1, \ldots, m$, v is the main successor of v_i, and*
- *all nodes $v_1, \ldots, v_m$ are labeled with the same n-ary function symbol f in G, and*
- *there exist states $q_1, \ldots, q_n$ such that for all successors $v_1^1, \ldots, v_1^n, \ldots, v_m^1, \ldots, v_m^n$ of $v_1, \ldots, v_m$ in the graph G, we have $q_i = r(v_1^i) = r(v_2^i) = \ldots = r(v_m^i)$ for $i = 1, \ldots, n$.*

Let $\langle {v'}_1^1, \ldots, {v'}_1^{k-1}, {v'}_1^{k+1}, \ldots, {v'}_1^n \rangle, \ldots, \langle {v'}_m^1, \ldots, {v'}_m^{k-1}, {v'}_m^{k+1}, \ldots, {v'}_m^n \rangle$ be the first (according to the lexicographic extension of $\prec$) m sequences of nodes providing the states $q_1, \ldots, q_{k-1}, q_{k+1}, \ldots, q_n$, where k is the position of v in the list of successors of v_i. The fork condition *is that the predecessor v_i is labeled with*

$$f(q_1[\mathsf{ind}_r({v'}_i^1)], \ldots, \underline{q_k}, \ldots, q_n[\mathsf{ind}_r({v'}_i^n)]) \to r(v_i).$$

Claim 1 *S is a skeleton.*

We have to prove that each used pointer points to some node below and that the induced graph is a t-dag (the last two conditions in Definition 11).

For the first of them, note that in the definition of S pointers $q_j[\mathsf{ind}_r(v'^j_i)]$ are used in labels of nodes v_i. Since $q_j[\mathsf{ind}_r(v'^j_i)]$ points to v'^j_i and $v'^j_i \prec v^j_i$, we have that v'^j_i lies below or at the same depth as v^j_i, which lies below v_i.

For the second condition, we have to show that for every pair G_1, G_2 of closed subgraphs of the graph induced from S, G_1 and G_2 are not isomorphic. Without loss of generality we can assume that G_1 and G_2 are t-dags, rooted at nodes v_1 and v_2 respectively. If v_1 and v_2 are of different depth then the graphs are obviously non-isomorphic, so suppose they are of the same depth. Now the proof goes by induction on this depth.

For the base case, if G_1 and G_2 are t-dags of depth 1, then they both consist of a single node labeled with a constant symbol. If v_1 is different from v_2, then the constants labeling them are different, and thus G_1 and G_2 are not isomorphic.

For the induction step, assume that no two graphs of depth n are isomorphic, and that v_1 and v_2 are nodes of depth $n+1$. Suppose that G_1 and G_2 are isomorphic. Then v_1 and v_2 must have the same main successor v (otherwise, from the induction assumption the corresponding subgraphs are non-isomorphic). If the main successor occurs on the same position (in case of a fork) then by construction the labels used at v_1 and v_2 in S are different and G_1 and G_2 cannot be isomorphic. Suppose v occurs on two different positions in the lists of successors of v_1 and v_2 in G. Then v_1 and v_2 are labeled in S with transitions of the form $f(q_1[i_1], \ldots, \underline{q_i}, \ldots, q_j[i_j], \ldots, q_n[i_n]) \to q$ and $f(q_1[i_1], \ldots, q_i[i_i], \ldots, \underline{q_j}, \ldots, q_n[i_n]) \to q'$, with $q_i = q_j$ and $i < j$. Suppose v_1 is labeled with the first one. By the definition of the main path (for v_2 in the graph G), the i-th successor of v_2 lies strictly below the node v and thus the graph rooted at $q_i[i_i]$ cannot be isomorphic with the graph rooted at v, which is required by the isomorphism between G_1 and G_2. $\square$

A node v in S is called a *milestone* if it is the root of G or there exist a pointer $q[i]$ used in a label in S, pointing to v.

Claim 2 *S contains at most $|Q|^2$ milestones.*

First note that each node in S is either a milestone of index 1 (this includes the root of G), or a milestone of index greater than 1, or lies on a main path for some milestone.

How many milestones do we need to obtain different labels for all predecessors of a fork of degree m? If $k_1, \ldots, k_n$ are the numbers of milestones providing the states $q_1, \ldots, q_n$, respectively, then we can construct $k_1 \cdot \ldots \cdot k_n$ different sequences of the form $\langle v'_1, \ldots, v'_n \rangle$ (cf. the fork condition). We need m such sequences, and since $k_1 \cdot \ldots \cdot k_n \geq (k_1 - 1) + \ldots + (k_n - 1) + 1$, it is enough if $(k_1 - 1) + \ldots + (k_n - 1) \geq m - 1$. Therefore a fork of degree m requires at most $m - 1$ milestones of index greater than 1.

The degree of a fork cannot exceed the number of nodes of the same depth in S, we call this number the *width* of S.

There are at most $|Q|$ milestones of index 1 (one per state). Hence, for each d there are at most $|Q|$ nodes of depth d lying on main paths for milestones of index 1. To estimate the width of S note that a milestone of index greater than 1 is introduced only to satisfy the fork condition for some fork. A fork of degree m at depth d is the only successor of m nodes of depth $d+1$ in S, and thus it decreases the width of S at depth d and below by $m-1$; on the other hand it may introduce at most $m-1$ milestones of index greater than 1 (and below them the nodes on main paths for them) at depth d or below, and thus it may increase the width of S at depth d or below by at most $m-1$. Hence, the width at each depth is bounded by $|Q|$.

Since the degree of every fork is bounded by the width of S (which is bounded by $|Q|$), each fork requires at most $|Q|-1$ milestones of index greater than 1. Thus the maximal index of a milestone does not exceed $|Q|$ and there are at most $|Q|^2$ milestones. □

A node which is neither milestone nor have more than one predecessor in S is called *ordinary.*

Claim 3 (pumping lemma) *Let the skeleton S contain a path $v_0 \to v_1 \to \ldots \to v_m$ such that*

- *the states provided by v_1 and v_m are the same, and*
- *all the nodes $v_1, \ldots, v_m$ are ordinary.*

If S' is a graph obtained from S by removing the nodes $v_1, \ldots, v_{m-1}$ and defining v_m as the immediate successor of v_0 then the graph induced from S' is a t-dag accepted by the automaton $\mathcal{A}$.

Note that S' need not be a skeleton, since it may contain nodes v with labels using pointers to nodes that do not lie below v (the last condition in Definition 11), but still it induces a directed acyclic graph G'. We have to show that G' is a t-dag and that $\mathcal{A}$ accepts it. The latter thing is quite simple: the mapping assigning to each node of G' the state provided by this node in S' is a successful run. Hence, it is enough to show that G' is a t-dag.

Suppose G' contains two different closed isomorphic subgraphs G'_1 and G'_2. If none of them contain the node v_m, then both G'_1 and G'_2 are closed subgraphs of G, which contradicts the fact that G is a t-dag. Suppose G'_1 contains v_m. Then G'_2 contains an isomorphic copy of the subgraph rooted at v_m, which must be a subgraph of G (the differences between G and G' start on the level above v_m). Since the only closed subgraph of G isomorphic to the subgraph rooted at v_m is the subgraph rooted at v_m, G'_2 also contains v_m. Hence the difference between G'_1 and G'_2 must occur somewhere above v_m. Since v_m is a node with only one predecessor v_0 in S' (and thus in G', too), both G'_1 and G'_2 contain v_0. Now let G_1 and G_2 be the two graphs obtained from G'_1 and G'_2 by replacing the edge $v_0 \to v_m$ with the path

$v_0 \to v_1 \to \ldots \to v_m$ together with all induced edges. Now the isomorphism between G'_1 and G'_2 can be extended to an isomorphism between G_1 and G_2, which contradicts the fact that G is a t-dag. □

Proof of Theorem 14. By repeated applications of the procedure above we can construct a t-dag H accepted by the automaton $\mathcal{A}$ such that the skeleton S_H constructed for H does not contain any path whose nodes are ordinary and two of them provide the same state. Thus every path consisting of ordinary nodes has the length bounded by $|Q|$. Every maximal path of this form has a unique predecessor which is not ordinary. Since each root (that is, a node without predecessors in S_H) is a milestone, the number of nodes that have more than one predecessor in S_H is bounded by the number of milestones. Therefore, there are at most $2|Q|^2$ nodes which are not ordinary, and the number of nodes in S_H (equal to the number of nodes in H) is bounded by $2|Q|^3$. □

3.4 Lower bound

Theorem 16 *The emptiness problem for t-dag automata is NP-hard.*

Proof. We will encode the 3-SAT problem, that is the problem whether a given propositional logic formula in conjunctive normal form where each conjunct is a clause consisting of exactly three literals, is satisfiable. It is well-known that 3-SAT is an NP-complete problem.

Let $C_1 \wedge \ldots \wedge C_n$ be an instance of the 3-SAT problem, where every clause C_i is a disjunction of the three literals $L_{i_1} \vee L_{i_2} \vee L_{i_3}$ and every literal $L_{i_j} \in \{x_1, \ldots, x_k, \neg x_1, \ldots, \neg x_k\}$.

Let Σ consist of $k+n$ constant symbols $a_1, \ldots, a_k, b_1, \ldots, b_n$, binary function symbol c and n-ary function symbol f. Let

$$Q = \{X_1, \ldots, X_k, X'_1, \ldots, X'_k, B_1, \ldots, B_n, C_1, \ldots, C_n, q\},$$

$F = \{q\}$, and

$$\begin{aligned}\Delta = \;& \{a_i \to X_i,\;\; a_i \to X'_i \mid i = 1, \ldots, k\} \\ \cup \;& \{b_i \to B_i \mid i = 1, \ldots, n\} \\ \cup \;& \{c(\varepsilon(L_{i_1}), B_i) \to C_i,\;\; c(\varepsilon(L_{i_2}), B_i) \to C_i,\;\; c(\varepsilon(L_{i_3}), B_i) \to C_i \mid \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad i = 1, \ldots, n\} \\ \cup \;& \{f(C_1, \ldots, C_n) \to q\}\end{aligned}$$

where $\varepsilon(L_{i_j}) = \begin{cases} X_{i_j} & \text{if } L_{i_j} = x_{i_j} \\ X'_{i_j} & \text{if } L_{i_j} = \neg x_{i_j} \end{cases}$.

Suppose that $C_1 \wedge \ldots \wedge C_n$ is satisfied by an assignment $\alpha : \{x_1 \ldots, x_n\} \to \{\mathit{true}, \mathit{false}\}$. Let L_i be the literal that makes the clause C_i true and let x_{j_i} be the variable used in the literal L_i. It is not difficult to see that $\mathcal{A} = \langle \Sigma, Q, F, \Delta \rangle$ accepts the t-dag representing the term $f(c(a_{j_1}, b_1), \ldots, c(a_{j_n}, b_n))$. The constants $b_1, \ldots, b_n$ are used here to make sure that the subgraphs corresponding to different clauses are not isomorphic.

Conversely, if there exists a t-dag G accepted by $\mathcal{A}$, then the assignment α given by $\alpha(x_i) = \mathit{true}$ iff the run assigns X_i (and not X_i') to the node representing a_i, satisfies $C_1 \wedge \ldots \wedge C_n$. □

Note that the signature Σ used in the proof above depends on the instance $C_1 \wedge \ldots \wedge C_n$. One can prove the same result for a fixed signature consisting of one constant and one binary symbol by replacing $a_1, \ldots, a_k, b_1, \ldots, b_n$ with $k+n$ different ground terms and constructing $\mathcal{A}$ in such a way that X_i and X_i' accept the t-dag representing the i-th of them, B_i accepts the $k+i$-th, and q is reached from a subgraph representing $c(C_1, c(C_2, \ldots, c(C_{n-1}, C_n) \ldots))$.

4 Directional type checking

In this section we prove that the directional type checking for logic programs wrt. to general regular types is decidable in NEXPTIME. This huge complexity does not seem to be a really big problem, since the types used in practice are usually not big (e.g. the automaton recognizing the set of lists has only two states). We do not have, however, an implementation of the method presented here.

If Σ is a signature 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.

A *type* T is a set of terms t closed under substitution [7]. 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 $t_1 : T_1 \wedge \ldots \wedge t_n : T_n \to t_0 : T_0$ that holds under 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 [5] or a regular grammar as in [21]). The definition below coincides with the types used in [5], it extends the definition from [21] by allowing non-ground types, and is equivalent to the definition from [13].

Definition 17 (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*

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

Definition 18 (Directional type of a program [14, 5]) *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 $p_0(t_0) \leftarrow p_1(t_1), \dots, p_n(t_n)$ of $\mathcal{P}$, the following type judgments hold.*

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

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

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 [23]. It is equivalent to other notions occurring in the literature: tuple-distributive [31, 34], discriminative [5], or deterministic.

Definition 19 (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 finite tree automaton translates to a logic program which does not contain two different clauses with the same head (modulo variable renaming), e.g., $p(f(x_1, \dots, x_n)) \leftarrow p_1(x_1), \dots, p_n(x_n)$ and $p(f(x_1, \dots, x_n)) \leftarrow p'_1(x_1), \dots, p'_n(x_n)$. A discriminative set expression as defined in [5] 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 [5] 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 $\{f(a, a), f(b, b)\}$. Other regular but not path-closed sets are for example lists of numbers containing exactly one element of a given type (which can be useful in reasoning about critical sections in infinite-state transition systems modeled by logic programs, cf. [20]) or 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).

We transform the well-typedness condition in Definition 18 into a logic program $\mathcal{P}_{InOut}$ by replacing $t : I_p$ with the atom $p^{In}(t)$ and $t : O_p$ with $p^{Out}(t)$. The following theorem is proved in [19]. Essentially, it says that 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.

Theorem 20 (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 $\mathcal{P}_{InOut}$.

The immediate consequence of this theorem is 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 $T_0, T_1, \ldots, T_n$, where $T_i = T_j$ whenever $p_i = p_j$, decide whether the set $\bigcup_{i=0}^{n}\{p_i(t) \mid t \in T_i\}$ is a model of the clause.*

To prove the decidability of this problem, we need the following lemma.

Lemma 21 *Let $\mathcal{A}_i = \langle \Sigma, Q_i, F_i, \Delta_i \rangle$ for $i = 0, \ldots, n$ be tree automata with disjoint sets of states, and let $g \notin \Sigma$ be a fresh function symbol of arity $n+1$. There exists a tree automaton $\mathcal{A} = \langle \Sigma \cup \{g\}, Q, F, \Delta \rangle$ such that*

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

Proof. By standard complementation and determinisation methods we construct an automaton $\mathcal{A}' = \langle \Sigma \cup \{g\}, Q', F', \Delta' \rangle$ 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 $\mathcal{A}'$.

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 $g(s_0, \ldots, s_n) \to s_{\text{fin}} \in \Delta'$ 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 each i, the automaton $\mathcal{A}'_i = \langle \Sigma \cup \{g\}, Q, \{s \in Q \mid s \cap F_i \neq \emptyset\}, \Delta \rangle$ recognizes exactly the set $\mathcal{L}(\mathcal{A}_i)$, and $\mathcal{A}'_0$ restricted to Σ is complete. $\square$

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 $T_0, T_1, \ldots, T_n$ be an instance of Problem 1. We did not specify here the formalism in which the sets $T_0, T_1, \ldots, T_n$ 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 [5] or regular grammars from [21] is straightforward.

The idea of the proof is to test the emptiness of the intersection of the automaton constructed in Lemma 21 with the set of instances of the term $g(t_0, \ldots, t_n)$. Due to non-linear occurrences of variables in $g(t_0, \ldots, t_n)$ this last set is, however, not regular. Automata on t-dags help us to handle these nonlinearities.

For each variable $x \in \mathsf{Var}(g(t_0, \ldots, t_n))$ we introduce a fresh constant symbol a_x. Let G be the t-dag representing the ground term $g(t_0, \ldots, t_n)\theta$ where θ is the substitution assigning to each variable $x \in \mathsf{Var}(g(t_0, \ldots, t_n))$ the respective constant a_x.

Let $\mathcal{A}_1 = \langle \Sigma, Q, F, \Delta_1 \rangle$ be a deterministic tree automaton without unreachable states, recognizing $g(T_{\Sigma - \{g\}} - T_0, T_1, \ldots, T_n)$, as constructed in Lemma 21.

Let $\mathcal{A}$ be a t-dag automaton $\langle \Sigma', Q, F, \Delta \rangle$ where $\Sigma' = \Sigma \cup \{a_x \mid x \in \mathsf{Var}(g(t_0, \ldots, t_n))\}$ and $\Delta = \Delta_1 \cup \{a_x \to q | q \in Q, x \in \mathsf{Var}(g(t_0, \ldots, t_n))\}$.

Claim 1 *The set $\bigcup_{i=0}^{n} \{p_i(t) \mid t \in T_i\}$ 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 the t-dag G 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 $\sigma : \mathsf{Var}(g(t_0, \ldots, t_n)) \to T_{\Sigma-\{g\}}$ such that $t_1\sigma \in T_1, \ldots, t_n\sigma \in T_n$ and $t_0\sigma \notin T_0$. This is equivalent to the existence of such a σ that the automaton $\mathcal{A}_1$ accepts the term $g(t_0, \ldots, t_n)\sigma$. Thus it enough to prove the equivalence of the last condition with the acceptance of G by $\mathcal{A}$. Recall that G is a t-dag representing $g(t_0, \ldots, t_n)\theta$.

Now we prove this equivalence. If $\mathcal{A}_1$ accepts $g(t_0, \ldots, t_n)\sigma$ then let q_x be the state assigned to the tree $x\sigma$. We assign q_x to the node labeled with a_x in G. Using a direct correspondence between $\mathcal{A}$ and $\mathcal{A}_1$ (as in Proposition 5; that is why $\mathcal{A}_1$ must be deterministic) we can extend this assignment to a successful run of $\mathcal{A}$ on G.

Conversely, if G is accepted by $\mathcal{A}$ then let q_x be the state assigned to the node labeled a_x in a successful run of $\mathcal{A}$. Since all states in Q are reachable (by the tree automaton $\mathcal{A}_1$), there exists a tree t_x accepted by the state q_x. Putting $\sigma(x) = t_x$ for all $x \in \mathsf{Var}(g(t_0, \ldots, t_n))$ we obtain a σ such that $\mathcal{A}_1$ accepts the term $g(t_0, \ldots, t_n)\sigma$. □

Theorem 22 *Problem 1 is decidable in NEXPTIME.*

Proof. This is a direct consequence of Proposition 7 and the claim above. □

The following corollary is a direct consequence of Theorems 20 and 22.

Corollary 23 *Directional type checking for logic programs wrt. arbitrary regular types is decidable in NEXPTIME.*

5 Automata on t-dags and set constraints

Decidability of the satisfiability problem for positive set constraints was first proved by Aiken and Wimmers [6] by syntactic transformations of set constraints. Three other proofs were given by Aiken, Kozen, Vardi and Wimmers [3] (techniques based on hypergraphs), Bachmair, Ganzinger and Waldmann [10] (techniques based on the equivalence between set constraints and monadic logic), and Gilleron, Tison and Tommasi [24] (automata-theoretic techniques). The results in [4, 15, 25] extend the respective techniques to prove decidability of the satisfiability problem for mixed positive and negative set constraints.

In this section we give another proof based on t-dag automata. We believe that it simplifies the three different approaches to solving negative set constraints [4, 15, 25].

Set constraints. Set constraints denote relations between sets of ground terms. They have numerous applications in program analysis and type inference (for pointers to the literature, see the overviews [2, 27, 32]).

Syntactically, a *positive* and *negative* set constraints are inclusions of the form $E \subseteq E'$ and $E \not\subseteq E'$ where the expressions E and E' are given by the grammar

$$E ::= X \mid \top \mid \bot \mid E \cup E \mid E \cap E \mid \overline{E} \mid f(E, \ldots, E)$$

where X stands for a variable from a given set, and f is a function symbol from a given signature Σ.

Semantically, the variables range over sets of ground terms from the Herbrand universe T_Σ over Σ, the symbols $\top$ and $\bot$ are interpreted as the T_Σ and the empty set, respectively, the Boolean connectives are interpreted in the usual way and

$$f(S_1, \ldots, S_n) = \{f(t_1, \ldots, t_n) \mid t_1 \in S_1, \ldots, t_n \in S_n\}.$$

A system SC of set constraints is satisfiable if there exists an assignment of subsets of T_Σ to the variables satisfying all the constraints in SC.

Let SC be a fixed system of set constraints. Define $E(SC)$ as the set of all expressions occurring in SC, except expressions of the form $\overline{E}$. For example, if SC is the constraint $f(\overline{X}, Y) \subseteq \overline{g(\top \cap X)}$ then

$$E(SC) = \{f(\overline{X}, Y), X, Y, g(\top \cap X), \top \cap X, \top\}.$$

Consider the set of Boolean functions $2^{E(SC)}$ from $E(SC)$ to $\{\mathit{true}, \mathit{false}\}$. Each such function φ can be seen as a set

$$\{E \mid \varphi(E) = \mathit{true}\} \cup \{\overline{E} \mid \varphi(E) = \mathit{false}\}.$$

If $\varphi(E) = \mathit{true}$ then we say that E occurs positively in φ, otherwise we say that E occurs negatively in φ.

Below we define an automaton, whose states are such Boolean functions. Intuitively, a state φ accepts the t-dags that represent ground terms which belong to all sets E occurring positively in φ and do not belong to all sets E occurring negatively in φ. For example, if $\varphi = \{\overline{f(\overline{X}, Y)}, \overline{X}, Y, g(\top \cap X), \overline{\top \cap X}, \top\}$ then the state φ should accept all the t-dags representing the terms from $\overline{f(\overline{X}, Y)} \cap \overline{X} \cap Y \cap g(\top \cap X) \cap \overline{\top \cap X} \cap \top$. The construction is essentially the same as in [24, 25], with some influence of the monadic formulas from [10].

The automaton corresponding to SC. Let $Q \subseteq 2^{E(SC)}$ be the set of Boolean functions $\varphi : E(SC) \to \{\mathit{true}, \mathit{false}\}$ such that

- $\varphi(SC) = \mathit{true}$, that is, if $E \subseteq E' \in SC$ and E occurs positively in φ then E' occurs positively in φ,
- if $\top \in E(SC)$ then $\top$ occurs positively in φ,

if $\perp \in E(SC)$ then $\perp$ occurs negatively in φ,

- if $E_1 \cup E_2$ occurs positively in φ then at least one of E_1, E_2 occurs positively in φ,

  if $E_1 \cup E_2$ occurs negatively in φ then both E_1, E_2 occur negatively in φ,

- if $E_1 \cap E_2$ occurs positively in φ then both E_1, E_2 occur positively in φ, and

  if $E_1 \cap E_2$ occurs negatively in φ then at least one of E_1, E_2 occurs negatively in φ.

Let Δ be the set of transitions of the form $f(\varphi_1, \ldots, \varphi_n) \to \varphi$ such that

- $f \in \Sigma$, the arity of f is n, and $\varphi_1, \ldots, \varphi_n, \varphi \in Q$,
- φ does not contain any positive occurrence of an expression of the form $g(E_1, \ldots, E_m)$ where $g \in \Sigma$ is a symbol different from f (this includes the case of g being a constant, if $m = 0$), and
- $f(E_1, \ldots, E_n)$ occurs positively in φ iff for all $i = 1, \ldots, n$, the expression E_i occurs positively in φ_i.

5.1 Positive set constraints

In case of positive set constraints we do not care about accepting states, so we define F as the empty set.

Below we consider a t-dag representation of the Herbrand universe, which is an infinite graph where each node corresponds to a ground term (more precisely, the closed subgraph rooted at this node represents the term).

An automaton $\langle \Sigma, Q, F, \Delta \rangle$ is a sub-automaton of $\langle \Sigma, Q', F', \Delta' \rangle$ if $Q \subseteq Q'$, $F \subseteq F'$ and $\Delta \subseteq \Delta'$.

Theorem 24 *Let SC be a system of positive set constraints, $\mathcal{A} = \langle \Sigma, Q, \emptyset, \Delta \rangle$ be the automaton corresponding to SC as defined above, and let H be the t-dag representing the Herbrand universe T_Σ. The following conditions are equivalent*

1. *SC is satisfiable*
2. *there exists a run of $\mathcal{A}$ on the t-dag H*
3. *the automaton $\mathcal{A}$ has a complete sub-automaton*

Sketch of the proof. First we consider the equivalence between 1 and 2. If SC is satisfied by a valuation α then the assignment of the set $\{E \in E(SC) \mid t \in \alpha(E)\} \cup \{\overline{E} \mid E \in E(SC),\ t \notin \alpha(E)\}$ to the node representing the term t defines a correct run of $\mathcal{A}$ on H. Conversely, if there exist a run of $\mathcal{A}$ on H then let $T(\varphi)$ be the set of the terms represented by the nodes accepted by the state φ. The assignment of the set $\bigcup\{T(\varphi) \mid X \text{ occurs positively in } \varphi\}$ to the variable X satisfies SC.

The equivalence between 2 and 3 is quite simple: if there exists a run r of $\mathcal{A}$ on H then $\mathcal{A}$ restricted to the states that are reachable in r (that is, to the image of r) defines the desired sub-automaton. Conversely, if $\mathcal{A}'$ is a closed sub-automaton of $\mathcal{A}$ then one can inductively define r on a node representing $f(t_1, \ldots, t_n)$ as the state q such that $f(q_1, \ldots, q_n) \to q$ is a transition of $\mathcal{A}'$ and r assigns q_i to the node representing t_i, for $i = 1, \ldots, n$. $\square$

Relations to the other approaches. Below we show a correspondence between this and other approaches to solving set constraints. In case of positive set constraints this correspondence gives a unified view on different techniques. In case of positive and negative set constraints, however, this correspondence together with our emptiness test simplifies quite significantly the proofs of the main results in corresponding papers.

A t-dag automaton can be seen as a hypergraph, where the states correspond to the nodes, and the transitions correspond to the hyperedges of the hypergraph. We believe that up to minor details, the construction of the t-dag automaton corresponding to a system of set constraints coincide with the analogous construction of the hypergraph corresponding to the same system of set constraints in [3, 4]. Under this correspondence, the equivalence between the conditions 1 and 3 coincide with Theorem 1 in [3] and Theorem 3 in [4].

A t-dag automaton behaves in the same way as a corresponding tree set automaton as defined in [24, 25]. The equivalence between the conditions 1 and 2 directly corresponds to Theorem 15 in [24], the equivalence between 2 and 3 to Theorem 4 in [24].

In [10], the authors prove that a system of positive set constraints is satisfiable iff the corresponding monadic formula is satisfiable. All predicates that occur in the formula are named P_E for some $E \in E(SC)$. If we replace the phrase "E occurs positively in φ" by "$P_E(\varphi)$" in the conjunction of the itemized conditions defining the automaton $\mathcal{A}$ above, then essentially we obtain the formula from [10], corresponding to SC. Now a complete sub-automaton $\mathcal{A}'$ of $\mathcal{A}$ can be seen as the model of this formula: states of the automaton are elements of the model and the interpretation of predicates are given by: $P_E(\varphi)$ holds iff E occurs positively in φ. The transitions $f(\varphi_1, \ldots, \varphi_n) \to \varphi$ give the interpretation of the skolem function for the formula in the model (if the formula is of the form $\forall x_1 \ldots \forall x_n \exists y \;\; \psi(x_1, \ldots, x_n, y)$ then the skolem function assigns to each sequence $x_1, \ldots, x_n$ such a y that $\psi(x_1, \ldots, x_n, y)$ holds). The sub-automaton must be complete to ensure that f is defined *for all* elements of the model. Thus, the theorem reducing satisfiability of set constraints to satisfiability of monadic formulas can be seen as the equivalence between conditions 1 and 3, together with a standard result on monadic logic (see [1], page 34) saying that a monadic formula with predicates from a given set P is satisfiable iff it has a model consisting of elements from 2^P.

5.2 Positive and negative set constraints

First note that a system of set constraints with n negative constraints can be reduced to an equivalent system with one negative constraint only. Simply note that $E \not\subseteq E'$ is equivalent to $E \cap \overline{E'} \not\subseteq \bot$ and replace $E_1 \subseteq E'_1, \ldots, E_n \subseteq E'_n$ with

$$f(E_1 \cap \overline{E'_1}, \ldots, f(E_{n-1} \cap \overline{E'_{n-1}}, E_n \cap \overline{E'_n}) \ldots) \not\subseteq \bot,$$

where f is a binary function symbol in Σ.

Let SC be a system of set constraints with one negative constraint $E \not\subseteq \bot$. Define the set of accepting states of the automaton corresponding to SC as

$$F = \{\varphi \in Q \mid E \text{ occurs positively in } \varphi\}.$$

Theorem 25 *Let $\mathcal{A} = \langle \Sigma, Q, F, \Delta \rangle$ be the automaton corresponding to SC as defined above. The following conditions are equivalent*

1. *SC is satisfiable*
2. *there exists a complete sub-automaton $\mathcal{A}'$ of the automaton $\mathcal{A}$ and a finite t-dag G accepted by $\mathcal{A}'$*

Proof. It is enough to note that in the proof of Theorem 24 above, the correspondence between a run of the automaton and a solution of set constraint is such that a final state is reachable iff the set assigned to E is nonempty. $\square$

Again there are strong relations between this theorem and the main theorems in the three papers showing decidability of the satisfiability problem for negative set constraints [4, 15, 25]. In [4], satisfiability for negative set constraints is reduced to another problem (Problem 4 in [4]), which translates (if SC contains one negative constraint) directly to the emptiness problem for a closed sub-automaton. The relation with [25] is also direct, since the satisfiability problem is there reduced to the emptiness problem of the corresponding tree set automaton. In case of [15], the emptiness problem is reduced to existence of a model of the corresponding formula (which translates to a complete sub-automaton) satisfying some property (essentially, existence of an accepted t-dag). Moreover, the technique we used here for testing emptiness is based on the technique in [15].

6 Conclusion and future work

We introduced automata on t-dags and proved the basic properties of the languages that they recognize: closure under union and intersection, NP-completeness of the emptiness problem, decidability in NP of the membership problem.

Using these automata, we proved the decidability of directional type checking of logic programs wrt. to general regular types. We also have shown another application of these automata to solving set-constraint problems.

There are several possible directions for future work. We did not answer the questions regarding determinisability of t-dag automata. We do not know whether the membership problem can be solved in deterministic polynomial time. We left a gap between the upper (NEXPTIME) and the lower (DEXPTIME) bounds for the directional type checking. It would also be interesting to have an implementation of the type check to see how it behaves in practice.

We plan to use the approach from Section 5 to simplify the proof of decidability of the satisfiability problem for set constraints with projections [16]. It should be also possible to extend these techniques to automata with equality tests [11] and use them for solving negative set constraints with equality [15].

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* North-Holland, Amsterdam, 1954.

[2] A. Aiken. Set constraints: Results, applications and future directions. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 326–335. Springer-Verlag, 1994.

[3] A. Aiken, D. Kozen, M. Vardi, and E. L. Wimmers. The complexity of set constraints. In *1993 Conference on Computer Science Logic*, LNCS 832, pages 1–17. Springer-Verlag, Sept. 1993.

[4] A. Aiken, D. Kozen, and E. L. Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, Oct. 1995.

[5] 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.

[6] A. Aiken and E. L. Wimmers. Solving systems of set constraints (extended abstract). In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, 1992.

[7] 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.

[8] 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.

[9] 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.

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

[11] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *9th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 577, pages 161–171, Paris, 1992. Springer-Verlag.

[12] F. Bossut, M. Dauchet, and B. Warin. Automata and rational expressions on planar graphs. In M. P. Chytil, L. Janiga, and V. Koubek, editors, *Mathematical Foundations of Computer Science 1988*, volume 324 of *lncs*, pages 190–200, Carlsbad, Czechoslovakia, 29 Aug.–2 Sept. 1988. Springer.

[13] J. Boye. *Directional Types in Logic Programming.* PhD thesis, Department of Computer and Information Science, Linköping University, 1996.

[14] 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.

[15] W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 128–136, 1994.

[16] W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In *Proceedings of the* 35^{th} *Symposium on Foundations of Computer Science*, pages 642–653, 1994.

[17] W. Charatonik and A. Podelski. Set constraints with intersection. In G. Winskel, editor, *Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 362–372. IEEE, June 1997.

[18] W. Charatonik and A. Podelski. Co-definite set constraints. In T. Nipkow, editor, *9th International Conference on Rewriting Techniques and Applications (RTA-98)*, LNCS 1379, pages 211–225. Springer-Verlag, 1998.

[19] 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.

[20] 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.

[21] 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.

[22] P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *LNCS*, Berlin, Germany, October 1997. Springer-Verlag.

[23] F. Gécseg and M. Steinby. *Tree Automata.* Akademiai Kiado, 1984.

[24] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *10th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 665, pages 505–514. Springer-Verlag, 1993.

[25] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proceedings of the* 34^{th}

Symp. on Foundations of Computer Science, pages 372–380, 1993. A full version *Technical report IT 247, Laboratoire d'Informatique Fondamentale de Lille.*

[26] R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. Technical Report IT 292, Laboratoire d'Informatique Fondamentale de Lille, 1996. 44 pages.

[27] 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.

[28] T. Kamimura and G. Slutzki. Dags and chomsky hierarchy. In H. A. Maurer, editor, *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, volume 71 of *LNCS*, pages 331–337, Graz, Austria, July 1979. Springer.

[29] T. Kamimura and G. Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, Apr. 1981.

[30] M. Kaminski and S. S. Pinter. Finite automata on directed graphs. *Journal of Computer and System Sciences*, 44(3):425–446, June 1992.

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

[32] L. Pacholski and A. Podelski. Set constraints - a pearl in research on constraints. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Springer LNCS*, Berlin, Germany, October 1997. Springer-Verlag.

[33] A. Potthoff, S. Seibert, and W. Thomas. Nondeterministic versus deterministic of finite automata over directed acyclic graphs. *Bulletin of the Belgian Mathematical Society - Simon Stevin*, 1:285–298, 1994.

[34] 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.

[35] W. Thomas. On logics, tilings, and automata. In J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *Lecture Notes in Computer Science*, pages 441–454, Madrid, Spain, 8–12 July 1991. Springer-Verlag.

[36] W. Thomas. Automata theory on trees and partial orders. In *Proc. 7th International Joint Conference CAAP/FASE: Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 20–34, Lille, France, 1997. Springer-Verlag, Berlin.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-99-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jaquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph
MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines

MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems
MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*
MPI-I-98-1-011	G.N. Frederickson, R. Solis-Oba	Robustness analysis in combinatorial optimization
MPI-I-98-1-010	R. Solis-Oba	2-Approximation algorithm for finding a spanning tree with maximum number of leaves
MPI-I-98-1-009	D. Frigioni, A. Marchetti-Spaccamela, U. Nanni	Fully dynamic shortest paths and negative cycle detection on diagraphs with Arbitrary Arc Weights
MPI-I-98-1-008	M. Jünger, S. Leipert, P. Mutzel	A Note on Computing a Maximal Planar Subgraph using PQ-Trees
MPI-I-98-1-007	A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr	On the Design of CGAL, the Computational Geometry Algorithms Library
MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values
MPI-I-97-2-009	A. Bockmayr, F. Eisenbrand	On the Chvátal Rank of Polytopes in the 0/1 Cube
MPI-I-97-2-008	A. Bockmayr, T. Kasper	A Unifying Framework for Integer and Finite Domain Constraint Programming
MPI-I-97-2-007	P. Blackburn, M. Tzakova	Two Hybrid Logics
MPI-I-97-2-006	S. Vorobyov	Third-order matching in $\lambda \rightarrow$-Curry is undecidable
MPI-I-97-2-005	L. Bachmair, H. Ganzinger	A Theory of Resolution