Paths vs. Trees in Set-based Program Analysis

Witold Charatonik[1,2] Andreas Podelski[1] Jean-Marc Talbot[1]

[1] Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbrücken, Germany
{witold, podelski, talbot}@mpi-sb.mpg.de
[2] University of Wrocław, Poland

Abstract

Set-based analysis of logic programs provides an accurate method for descriptive type-checking of logic programs. The key idea of this method is to upper approximate the least model of the program by a regular set of trees. In 1991, Frühwirth, Shapiro, Vardi and Yardeni raised the question whether it can be more efficient to use the domain of sets of paths instead, *i.e.*, to approximate the least model by a regular set of words. We answer the question negatively by showing that type-checking for path-based analysis is as hard as the set-based one, that is DEXPTIME-complete. This result has consequences also in the areas of set constraints, automata theory and model checking.

1 Introduction

Type-checking for path-based approximation. Descriptive types for logic programs are defined as a conservative approximation of the least model of a program. Type inference is performed automatically without any additional information from the programmer. Such type information can be then used for debugging of the program or for optimizations during its compilation. The type-checking problem is to determine whether a ground atom belongs to such conservative approximation. An atom belonging to the approximation is said to be well-typed, otherwise it is said to be ill-typed. This view of typing is optimistic in the sense that well-typed atoms may succeed for the program, whereas ill-typed atoms certainly do not succeed.

In [11], Frühwirth, Shapiro, Vardi and Yardeni proposed to represent those approximations, and thus types, as syntactically restricted logic programs. These restrictions are necessary for the decidability of the type-checking problem. The type inference mechanism is then given by a syntactic transformation of a logic program into a "simpler" one, so-called proper unary-predicate program. The notion of types it defines coincides with the one proposed by Heintze and Jaffar in [14], called set-based analysis. The basis of this analysis is to present types as sets of trees.

POPL 2000 Boston MA USA

The program P below is a proper unary-predicate program, and thus its set-based approximation is exact — it coincides with the least model of the program.

$$\begin{array}{l} p_1(f(a,b)) \\ p_2(f(a,a)) \\ p_3(f(b,b)) \\ q(x) \leftarrow p_1(x), p_2(x) \\ q(x) \leftarrow p_1(x), p_3(x) \end{array}$$

On can notice that the ground atom $p_1(f(a,b))$ is well-typed wrt. P since it belongs to the least model of the program. Moreover, for any ground term t, $q(t)$ is ill-typed since the denotation of the predicate q in the program P is empty.

Frühwirth, Shapiro, Vardi and Yardeni showed that the type-checking problem for the set-based approximation (*i.e.* the membership problem of a ground atom in the least model of a proper unary-predicate program) is DEXPTIME-complete.

There is another conservative approximation mentioned in [11], namely the path-based approximation.[1] It has already been considered in [10, 23] and is rephrased in [11] in terms of program transformation. When set-based approximation is based on sets of trees, path-based approximation is based on sets of words: the key idea of the path-based approximation is to view atoms (or sets of atoms) as sets of paths (which are sets of words) occurring in these atoms. Function symbols f of arity n strictly greater than 1 are replaced by n unary function symbols $f_1, \ldots, f_n$. The term $f(a,b)$ is then considered as the sets of paths $\{f_1(a), f_2(b)\}$ and the set of terms $\{f(a,b), f(b,b)\}$ as the set $\{f_1(a), f_2(b), f_1(b)\}$.

The path-based approximation is presented in [11] in terms of a transformation of a logic program into a unary one, where both predicate and function symbols are at most unary. This approximation is rougher than the set-based one, as illustrated with the example below: from the program P above, one obtains the following program π_P, whose semantics is the path-based approximation of P.

$$\begin{array}{ll} p_1(f_1(a)) & p_1(f_2(b)) \\ p_2(f_1(a)) & p_2(f_2(a)) \\ p_3(f_1(b)) & p_3(f_2(b)) \\ q(x) \leftarrow p_1(x), p_2(x) & \\ q(x) \leftarrow p_1(x), p_3(x) & \end{array}$$

[1]The path-based approximation is often confused with the *path-closed* one, for which the decidability of type checking is not known. See Appendix for a discussion.

The type-checking problem is formulated here as the inclusion of the finite set of paths of a ground atom in the path-based approximation of the program. For instance, the least model of π_P contains the two atoms $q(f_1(a))$ and $q(f_2(b))$. Since the ground atom $q(f(a,b))$ which is ill-typed for the (set-based approximation of) program P, is identified with the set of paths $\{q(f_1(a)), q(f_2(b))\}$, it is well-typed for the path-based approximation π_P of P.

Since the path-based approximation of a unary logic program coincides with the program itself, the type-checking problem for the path-based approximation is equivalent to the membership problem of a ground atom in the least model of a unary logic program.

As path-based approximation is strictly "weaker" than the set-based one, Frühwirth, Shapiro, Vardi and Yardeni raised the question whether the type-checking problem for the path-based approximation would be simpler than for the set-based approximation. We answer negatively this question by showing that the type-checking problem for the path-based approximation is also DEXPTIME-complete.

Set constraints. Set constraints denote relations between sets of trees. Syntactically, they are conjunctions of inclusions between expressions built over variables, constructors (constants and function symbols from a given alphabet) and a choice of set operators that defines the specific class of set constraints. Their main application domain is set-based program analysis and type inference for functional [20], imperative [17] and logic programming [14] languages, but they are also used in order-sorted languages and in constraint logic programming. See [1, 16, 19] for overviews of this area.

Definite set constraints were introduced by Heintze and Jaffar in [13] and used in set-based program analysis. This class is strictly less expressive than the general class of set constraints (where all boolean set operators, that is, union, intersection and complement, as well as function symbols of arbitrary arity are allowed). In fact, it was observed in [7] that definite set constraints are exactly as expressive as set constraints with intersection as the only boolean operator, and that the satisfiability problem for constraints in this class is DEXPTIME-complete.

Unary set constraints were first introduced by Aiken, Kozen, Vardi and Wimmers in [2]. This is a class of set constraints that does not allow the use of function symbols of arity greater than 1. Obviously, this class is also strictly less expressive than the general one. It is shown in [2] that the satisfiability problem for unary set constraints is DEXPTIME-complete, as opposed to the NEXPTIME-completeness of the non-unary case. The DEXPTIME-hardness proof presented there depends essentially on the complement operator, which is not allowed in definite constraints.

In both cases (definite or unary) the restriction in expressivity decreases the complexity of the satisfiability problem (see the right part of Figure 1). A similar decrease of complexity (to PSPACE) was expected if one takes the both restrictions at the same time. However, as we prove below, this is not the case: the satisfiability problem for unary definite set constraints is DEXPTIME-complete.

There are quite strong connections between definite or unary set constraints and tree automata. In particular, the DEXPTIME-hardness result for definite set constraints works by an encoding of the intersection emptiness problem for tree automata. Unary set constraints are equivalent to the existential fragment of the second-order theory of n successors, and thus can be solved using tree-automata techniques [22]. Since for many DEXPTIME-complete problems in the tree-automata theory (like intersection emptiness, language equivalence, universality) their corresponding problems for string automata are PSPACE-complete, it was expected that satisfiability of unary definite set constraints should also be PSPACE-complete. Surprisingly, as we show here, this problem is DEXPTIME-complete.

Automata theory. One can think of a nondeterministic finite automaton (NFA) as a machine equipped with a finite control and a stack, where the input word is written on the stack and the automaton is allowed to do only pop operations (it is not allowed to push anything on the stack). Büchi [4] extended word automata to canonical systems, which may be viewed as finite automata with both pop and push operations. It is well-known that the emptiness problem for such automata (often called pushdown processes) is decidable in PTIME. Another possible extension of finite automata is alternation. The emptiness problem for alternating word automata is PSPACE-complete. A natural question that arises here is what happens if we add both: alternation and push operations (see the left part of Figure 1). As we show in this paper, already the membership problem (and thus, also emptiness) for such automata is DEXPTIME-hard (the membership in DEXPTIME is easy by reduction to the tree case, so these problems are DEXPTIME-complete). These are the first problems that we are aware of that concern finite automata on words and are DEXPTIME-complete. It is worth noting here that a similar extension for alternating tree automata does not change the complexity (DEXPTIME) of the emptiness problem [11, 9, 6].

One should not confuse the automata considered here with alternating pushdown automata from [5, 18]. The difference is that alternating pushdown automata have not only stack, but also input tape, which gives them much more computational power: while alternating automata with push operations recognize only regular languages, the class of languages accepted by alternating pushdown automata is $\bigcup_{c>0} \mathrm{DTIME}(c^n)$.

Model checking. Bouajjani, Esparza and Maler in [3] considered a problem of model checking for pushdown systems. They reduce this problem to a reachability problem for what they call alternating pushdown systems and show that this problem is decidable in DEXPTIME. These alternating pushdown systems are exactly the above mentioned finite automata extended with push operations and alternation. The problem whether a given alternating pushdown systems accepts a given word is a particular case of the reachability problem. This shows that the reachability algorithm from [3] is optimal.

2 Preliminaries

Unary definite set constraints. The class of definite set constraints was the first class of set constraints for which decidability was shown [13, 15]. It was introduced by Heintze and Jaffar and is used for the type analysis of Prolog programs [14, 12, 16]. The satisfiable constraints in this class have a least solution (this fact is at the origin of the attribute "definite").

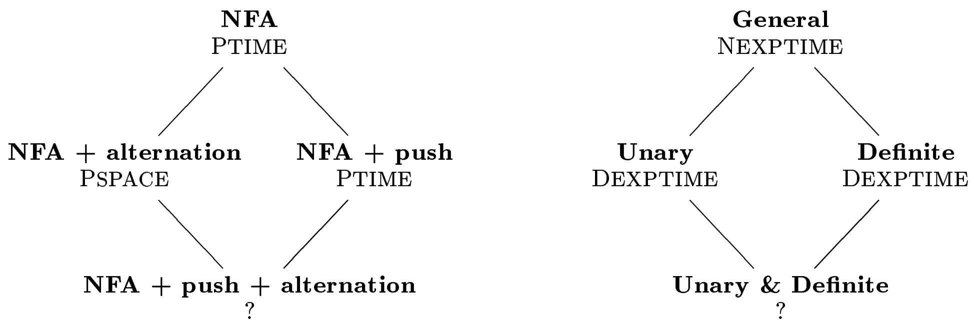

Figure 1: Complexity for problems in automata theory and set constraints

Formally, we define the left and right set expressions and unary definite set constraints by the following grammar[2]

$$\begin{array}{lcl} le & ::= & x \mid c \mid f(le) \mid f^{-1}(le) \mid le \cup le \mid le \cap le \mid \top \\ re & ::= & x \mid c \mid f(re) \\ sc & ::= & le \subseteq re \mid sc \wedge sc \end{array}$$

where c and f range over constants and unary function symbols from a given signature Σ, and x ranges over the set Var of variables. Note that there is no complement operator here.[3]

A solution of a set constraint is a valuation $\alpha : \mathsf{Var} \to 2^{T_\Sigma}$ that assigns to variables sets of ground terms over Σ. In our case, since Σ does not contain symbols of arity greater than 1, T_Σ is set of strings rather than trees. A valuation α is a solution of a set constraint, if for every conjunct $le \subseteq re$ we have $\alpha(le) \subseteq \alpha(re)$, where

$$\begin{array}{rcl} \alpha(a) & = & \{a\} \\ \alpha(f(exp)) & = & \{f(t) \mid t \in \alpha(exp)\} \\ \alpha(f^{-1}(exp)) & = & \{t \mid f(t) \in \alpha(exp)\} \\ \alpha(exp \cup exp') & = & \alpha(exp) \cup \alpha(exp') \\ \alpha(exp \cap exp') & = & \alpha(exp) \cap \alpha(exp') \\ \alpha(\top) & = & T_\Sigma \end{array}$$

The satisfiability problem for set constraints is the problem of deciding whether there exists a solution for a given set constraint.

Alternating pushdown systems. Alternating pushdown systems were introduced by Bouajjani, Esparza and Maler in [3] in a context of model-checking of pushdown systems.

Formally, an alternating pushdown systems is a tuple $\langle Q, \Sigma, \bot, \Delta\rangle$, where Q is a finite set of states, Σ is a finite stack alphabet, $\bot \in \Sigma$ is a bottom stack symbol, and Δ is a function that assigns to each element of $Q \times \Sigma$ a positive (that is, negation-free) boolean formula over elements of $Q \times \Sigma^*$. We assume that the symbol $\bot$ can be neither put nor removed from stack, that is, Δ assigns to elements from $Q \times (\Sigma - \{\bot\})$ formulas over $Q \times (\Sigma - \{\bot\})^*$ and to the elements from $Q \times \{\bot\}$ formulas over $Q \times (\Sigma - \{\bot\})^* \bot$. For better readability we will assume that these boolean formulas are in disjunctive normal form, which allows to define Δ equivalently as a subset of the set of transition rules $(Q \times \Sigma) \times 2^{Q \times \Sigma^*}$. By convention, the formula *true* is identified with the empty set. For example, instead of

$$\Delta(q, a) = ((q_1, w_1) \vee (q_2, w_2)) \wedge (q_3, w_3)$$

we write

$$\left\{ \begin{array}{l} \langle (q, a), \{(q_1, w_1), (q_3, w_3)\}\rangle, \\ \langle (q, a), \{(q_2, w_2), (q_3, w_3)\}\rangle \end{array} \right\} \subseteq \Delta$$

or simply $\begin{array}{l} (q, a) \to \{(q_1, w_1), (q_3, w_3)\} \\ (q, a) \to \{(q_2, w_2), (q_3, w_3)\} \end{array}$

and instead of

$$\Delta(q, a) = true$$

we write $\{\langle (q, a), \emptyset\rangle\} \subseteq \Delta$ or simply $(q, a) \to true$.

A configuration in a pushdown system $\langle Q, \Sigma, \bot, \Delta\rangle$ is a constant *true* or any pair (q, w) where $q \in Q$ and $w \in (\Sigma - \{\bot\})^* \bot$. Intuitively, in a configuration (q, aw) the system chooses nondeterministically a transition $(q, a) \to \{(q_1, w_1), \ldots, (q_n, w_n)\}$ (or $(q, a), \to true$; in this case it moves to the configuration *true*) from Δ and splits into n copies. Then, the i-th copy pops a from stack, pushes there w_i and moves to the state q_i, that is, the i-th copy moves from the configuration (q, aw) to $(q_i, w_i w)$. In a configuration *true* the system remains forever.

The reachability relation $\Rightarrow$ between configurations and sets of configurations in a pushdown system is defined inductively as follows.

- $(q, aw) \Rightarrow \{(q_1, w_1 w), \ldots, (q_n, w_n w)\}$ for all $(q, a) \to \{(q_1, w_1), \ldots, (q_n, w_n)\} \in \Delta$
- $(q, aw) \Rightarrow \{true\}$ if $(q, a) \to true \in \Delta$
- $c \Rightarrow \{c\}$ for all configurations c
- if $c \Rightarrow \{c_1, \ldots, c_k\}$ and $c_i \Rightarrow C_i$ for all $1 \le i \le k$ then $c \Rightarrow C_1 \cup \ldots \cup C_k$

[2] This grammar comes directly from restricting definite set constraints as introduced by Heinze and Jaffar in [13] to the unary signature. However, using an observation from [7] we can remove union and projection from the left-hand side or add intersection on the right-hand side without changing the expressivity. In the unary case the expressivity also does not change if we add projection on the right-hand side.

[3] One can express complement using union and intersection. Definite set constraints are equally expressive as set constraints with intersection as the only boolean set operator. In the unary case these constraints are also equally expressive (by duality) to the set constraints with union as the only boolean set operator. Since we do not have both union and intersection at the same time, we cannot express complement.

The reachability problem for alternating pushdown systems is the following problem. Given alternating pushdown systems $\mathcal{P}$, a configuration c and [a regular] set of configurations C, does c reach [a subset of] C in $\mathcal{P}$?

A particular case of the reachability problem is the following acceptance problem. Given alternating pushdown systems $\mathcal{P}$ and a configuration c, does c reach $\{true\}$ in $\mathcal{P}$? The main result of this paper is that the acceptance is DEXPTIME-hard, which shows that the reachability cannot be decided faster than in exponential time.[4]

Alternating Turing machines. Alternating Turing machines (ATM) were introduced by Chandra, Kozen and Stockmeyer in [5]. They generalize nondeterministic Turing machines in the same way as alternating finite automata generalize finite automata. Formally, an alternating Turing machine M is a tuple $\langle Q, \Sigma_{\mathsf{In}}, \Sigma, \Delta, q_0, \flat, F, U\rangle$, where

- Q is a finite set of states,
- Σ_{In} is a subset of Σ called the set of input symbols,
- Σ is a finite set of tape symbols,
- $\Delta : Q \times \Sigma \to 2^{Q\times\Sigma\times\{\mathsf{left},\mathsf{right}\}}$ is the transition function,
- q_0 is the initial state of M,
- $\flat$ is a symbol in Σ called blank,
- $F \subseteq Q$ is the set of final states, and
- $U \subseteq Q$ is the set of universal states.

A configuration of M is a string of the form vqw where $q \in Q$ and $vw \in \Sigma^*$. The position of q in vqw marks the position of the head of M on the tape containing the word vw (the machine reads the first symbol of w).

A configuration of the form q_0w is called initial. A configuration vqw is called *final* if $q \in F$, *universal* if $q \in U$, and *existential* if $q \notin U$. A configuration c' is a successor of a non-final configuration c, in symbols $c \vdash c'$, if c' follows from c in one step, according to the transition function Δ.

As an illustration, let us consider a configuration $vbqaw$ with $v, w \in \Sigma^*$, $a, b \in \Sigma$ and $q \in Q$. For a transition function Δ associating the set $\{(q', a', \mathsf{left}), (q'', a'', \mathsf{right})\}$ with (q, a), the two configurations $vq'ba'w$ and $vba''q''w$ are (the unique) successors of $vbqaw$.

We say that a configuration c *leads to acceptance* if

- c is final, or
- c is existential and there exist a successor of c that leads to acceptance, or
- c is universal and every successor of c leads to acceptance.

A word w is accepted by M if the initial configuration q_0w leads to acceptance. Equivalently, a word w is accepted iff there exists an *accepting computation tree* for M and w, that is a tree whose nodes are labeled with configurations, such that the root is labeled with the initial configuration; each intermediate node labeled with an existential configuration has one child labeled with a successor configuration; each intermediate node labeled with a universal configuration has children labeled with all successor configurations; and all leaves are labeled with final configurations.

Note that the acceptance condition for an ATM with an empty set of universal states is the same as the usual acceptance condition for Turing machines.

The proof of our main result is based on the result from [5] saying that the class of problems that can be solved in alternating polynomial space is the same as the class of problems solvable in deterministic exponential time (APSPACE=DEXPTIME)

Unary logic programs. We will use logic programs to encode computations of alternating Turing machines. The programs that we use are usual logic programs, with several restrictions. The most important restriction is that all predicate and all function symbols (except one constant symbol) are unary. Less important is that we allow only one variable and only flat terms (that is, terms of depth at most one) in heads of clauses. Formally, let Σ be a set of function symbols consisting of one constant symbol $\perp$ and finitely many unary symbols $f, g, \ldots$, let Pred be a finite set of predicate symbols, and let x be a variable. A unary logic program is a finite set of clauses of the form

$$p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$$

or $p_0(t_0) \leftarrow true$, where $p_0, \ldots, p_n \in \mathsf{Pred}$, $t_0, \ldots, t_n$ are terms over $\Sigma \cup \{x\}$ with t_0 being flat, that is, t_0 is either $\perp$ or x or $f(x)$ for some $f \in \Sigma$, with an additional restriction that all clauses with $p_0(\perp)$ in the head have *true* in the body.

A proof tree for a ground atom A and a logic program $\mathcal{P}$ is a tree whose nodes are labeled with ground atoms, such that the root is labeled with A; for each intermediate node that is labeled with B and has children labeled $B_1, \ldots, B_n$, the clause $B \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in $\mathcal{P}$; and all leaves are labeled *true*.

We say that an atom A belongs to the least model of a program $\mathcal{P}$ if there exists a proof tree for A and $\mathcal{P}$.

The membership problem for such programs is to decide, whether a given ground atom $p(t)$ belongs to the least model of a given program $\mathcal{P}$.

It is immediate to see that the membership problem for unary logic programs reduces to the acceptance problem for alternating pushdown systems. A unary logic program over Σ rewrites directly to an alternating pushdown system $\langle \mathsf{Pred}, \Sigma - \{\perp\}, \perp, \Delta\rangle$, where Δ is obtained from the set of clauses roughly by reversing the arrows and removing the variable x. The only detail here is that while rewriting a clause of the form

$$p_0(x) \leftarrow p_1(t_1(x)), \ldots, p_n(t_n(x))$$

one should first replace it with the set of clauses

$$p_0(f(x)) \leftarrow p_1(t_1(f(x))), \ldots, p_n(t_n(f(x)))$$

for all unary $f \in \Sigma$ together with a clause $p_0(\perp) \leftarrow p_1(t_1(\perp)), \ldots, p_n(t_n(\perp))$.

It is quite easy to see that the membership problem reduces to the satisfiability problem for unary definite set constraints. To see this, fix a program $\mathcal{P}$, and construct the following set constraint φ. The set variables that occur in φ are exactly these elements of Pred that occur in

[4] [3] contains a misleading statement that the reachability problem can be solved in time polynomial in the size of $\mathcal{P}$ and exponential in the size of [the description of] C. As we show here, this is not true in general (in the case of our acceptance problem the size of C is constant). It holds for the restricted case considered in [3], where the size of C is greater than the size of $\mathcal{P}$.

$\mathcal{P}$. For each clause $p(t) \leftarrow \mathit{true}$ the constraint φ contains the inclusion $t' \subseteq p$, where t' is the expression obtained from t by replacing the variable x with the symbol $\top$. For a given term $t(x)$ let t^{-1} be a context obtained from $t(\cdot)$ by reversing the order of symbols and adding to each symbol the superscript -1, for example, if $t(x) = f(g(h(x)))$ then $t^{-1} = h^{-1}(g^{-1}(f^{-1}(\cdot)))$. For each clause $p_0(f(x)) \leftarrow p_1(t_1(x)), \ldots, p_n(t_n(x))$ in $\mathcal{P}$ the constraint φ contains the inclusion $f(t_1^{-1}(p_1) \cap \ldots \cap t_n^{-1}(p_n)) \subseteq p_0$. For the clauses $p_0(x) \leftarrow \ldots$ we just omit the symbol f on the left-hand side of the corresponding inclusion.

It is quite obvious that the least model of the program and the least solution of φ represent the same sets: a clause $p_0(t_0) \leftarrow p_1(t_1), \ldots, p_n(t_n)$ says that the set p_0 contains all the terms of the form t_0 if t_i is a member of p_i for all $i = 1, \ldots, n$. Exactly the same condition is expressed by the corresponding conjunct in φ. Now the answer to an instance $\mathcal{P}, p(t)$ of the membership problem is 'no' if and only if the constraint $\varphi \wedge p \cap t \subseteq \bot \wedge p \cap t \subseteq f(\bot)$ is satisfiable. Note that the two conjuncts $p \cap t \subseteq \bot$ and $p \cap t \subseteq f(\bot)$ express simply that the set $p \cap t$ is empty.

It is not difficult to present reductions in the other directions, that is, for a given set constraint or a given pushdown system one can find a corresponding unary logic program. However, for the lower bounds for pushdown systems and set constraints, these reductions are not relevant. The exponential upper bounds for these problems are proved in [3, 7].

3 The main result

In this section we present our main result, that is, DEXPTIME-hardness of the membership problem for unary logic programs. By the reductions presented above, it also shows the DEXPTIME-hardness of the reachability problem for alternating pushdown systems and of the satisfiability problem for unary definite set constraints.

Fix an alternating Turing machine $M = \langle Q, \Sigma_{\mathsf{In}}, \Sigma, \Delta, q_0, \flat, F, U\rangle$ working in polynomial space and a word w. Let P be a polynomial such that M uses at most $P(|w|)$ tape cells, and let $n = P(|w|) + 2$ (we will use words of length n to encode configurations of M; for this we need $P(|w|)$ symbols to encode the tape content, one symbol for the state and one symbol to separate the current configuration from the next one).

We will write a unary logic program $\mathcal{P}_M$ and ground goal $p(t)$, both of size polynomial in the size of M and w, such that $p(t)$ is a member of the least model of $\mathcal{P}_M$ if and only if M accepts w.

The main idea to recognize whether the initial configuration leads to acceptance is to consider computation paths. A computation path for M is a word of the form $w_0\#w_1\#\ldots w_k\#$ such that for all $i < k$, w_{i+1} is a successor configuration of w_i. Such a path leads to acceptance if its last configuration w_k does. The program below checks this in the following way. If w_k is a final configuration, nothing has to be done. If w_k is existential, the program nondeterministically guesses a successor w_{k+1} and recursively checks that $w_0\#\ldots w_{k+1}\#$ leads to acceptance. If w_k is universal, the same is checked for all possible successors w_{k+1}.

First we fix the signature that we use, namely the set $\Sigma_{\mathcal{P}} = \Sigma \cup Q \cup \{\#, \bot\}$. We assume that neither $\#$ nor $\bot$ is a member of $\Sigma \cup Q$. All symbols except $\bot$ are unary, while $\bot$ is the only constant symbol.The symbol $\#$ will be used to separate configurations from each other.

From now on we identify words from $(\Sigma \cup Q \cup \{\#\})^*$ with ground terms over $\Sigma_{\mathcal{P}}$. For example a word abq_0c is identified with the term $c(q_0(b(a(\bot))))$. Note that the term representation of a word is reversed: adding a symbol at the end of a word results in adding a symbol in the beginning of the term.

Our program uses the following set of predicate symbols

$$\begin{array}{rl} \mathsf{Pred} = & \{\mathsf{leads\_to\_acc}, \mathsf{final}\} \\ \cup & \{\mathsf{check}_{i,\sigma} \mid \sigma \in \Sigma_{\mathcal{P}} - \{\bot\}, \ \ i = 0, \ldots, n-1\} \\ \cup & \{\mathsf{reads}_{q,a} \mid a \in \Sigma, \ \ q \in Q\} \\ \cup & \{\mathsf{succ}_\delta \mid \delta \in Q \times \Sigma \times Q \times \Sigma \times \{\mathsf{left}, \mathsf{right}\}\} \\ \cup & \{\mathsf{pushleft}_\delta \mid \delta \in Q \times \Sigma \times Q \times \Sigma \times \{\mathsf{left}, \mathsf{right}\}\} \\ \cup & \{\mathsf{trans}_\delta \mid \delta \in Q \times \Sigma \times Q \times \Sigma \times \{\mathsf{left}, \mathsf{right}\}\} \\ \cup & \{\mathsf{pushright}\}. \end{array}$$

Next we define the meaning of these predicates. The predicate final holds for all words from $(\Sigma_{\mathcal{P}} - \{\bot\})^* q \Sigma^*$ where $q \in F$; in particular it holds for all words that encode computation paths with the last configuration being final.

$$\begin{array}{ll} \mathsf{final}(q(x)) \leftarrow \mathit{true} & \text{for all } q \in F \\ \mathsf{final}(a(x)) \leftarrow \mathsf{final}(x) & \text{for all } a \in \Sigma \end{array}$$

The predicate $\mathsf{leads\_to\_acc}$ is defined by the clause

$$\mathsf{leads\_to\_acc}(\#(x)) \leftarrow \mathsf{final}(x),$$

the clauses

$$\mathsf{leads\_to\_acc}(\#(x)) \leftarrow \mathsf{reads}_{q,a}(x), \mathsf{succ}_{\langle q,a,q',a',m\rangle}(\#(x))$$

for all existential $q \in Q$, all $a \in \Sigma$ and all $\langle q', a', m\rangle \in \Delta(q, a)$, and the clauses

$$\mathsf{leads\_to\_acc}(\#(x)) \leftarrow \mathsf{reads}_{q,a}(x), \bigwedge_{\langle q',a',m\rangle \in \Delta(q,a)} \mathsf{succ}_{\langle q,a,q',a',m\rangle}(\#(x))$$

for all universal $q \in Q$ and all $a \in \Sigma$.

The predicate $\mathsf{reads}_{q,a}(x)$ says that in the last configuration in the path x, the machine M is in the state q, reading the tape symbol a. Formally it holds for all words in $(\Sigma \cup Q \cup \{\#\})^* qa\Sigma^*$. It is defined by

$$\begin{array}{ll} \mathsf{reads}_{q,a}(b(x)) \leftarrow \mathsf{reads}_{q,a}(x) & \text{for all } b \in \Sigma \\ \mathsf{reads}_{q,a}(a(x)) \leftarrow \mathsf{check}_{0,q}(x). & \end{array}$$

For all $i \leq n$ and all $\sigma \in \Sigma_{\mathcal{P}} - \{\bot\}$ we have a predicate $\mathsf{check}_{i,\sigma}$. The predicates $\mathsf{check}_{i,\sigma}$ hold for all terms encoding the words in $(\Sigma \cup Q \cup \{\#\})^* \sigma (\Sigma \cup Q \cup \{\#\})^i$. These predicates are defined by

$$\mathsf{check}_{i+1,\sigma}(\sigma'(x)) \leftarrow \mathsf{check}_{i,\sigma}(x)$$

for all $i \geq 0$, $\sigma, \sigma' \in \Sigma_{\mathcal{P}} - \{\bot\}$ and

$$\mathsf{check}_{0,\sigma}(\sigma(x)) \leftarrow \mathit{true}$$

for all $\sigma \in \Sigma_{\mathcal{P}} - \{\bot\}$

The predicate succ_δ holds for words of the form $uw\#$ such that w is a configuration whose successor by the transition δ leads to acceptance. This predicate is defined by a single clause

$$\mathsf{succ}_\delta(x) \leftarrow \mathsf{pushleft}_\delta(x).$$

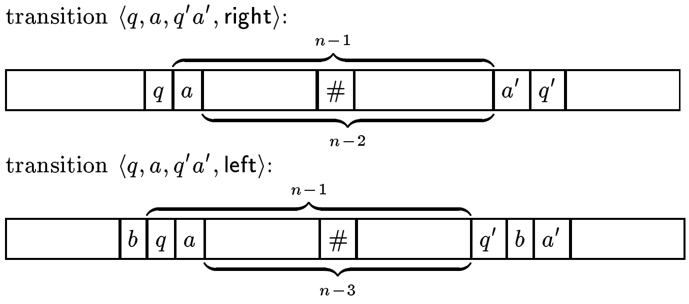

Figure 2: Transitions of a Turing machine

The predicate $\mathsf{pushleft}_\delta$ holds for words of the form $uw\#w'$ such that w is a configuration whose successor $uw\#w'w''$ by the transition δ leads to acceptance, where $w' \in \Sigma^*$. Operationally, this predicate is responsible for guessing a correct prefix of the successor configuration, up to the position of the head of the machine. It is defined by

$$\mathsf{pushleft}_\delta(x) \leftarrow \mathsf{pushleft}_\delta(a(x)), \mathsf{check}_{n-1,a}(x) \quad \text{for all } a \in \Sigma$$
$$\mathsf{pushleft}_\delta(x) \leftarrow \mathsf{trans}_\delta(x).$$

The predicate trans_δ is defined depending on the move of the head of the Turing machine, for $\delta = \langle q, a, q'a', \mathsf{right}\rangle$ by the clause

$$\mathsf{trans}_{\langle q,a,q'a',\mathsf{right}\rangle}(x) \leftarrow \quad \mathsf{pushright}(q'(a'(x))), \mathsf{check}_{n-1,q}(x), \mathsf{check}_{n-2,a}(x)$$

and for $\delta = \langle q, a, q'a', \mathsf{left}\rangle$ by the clauses

$$\mathsf{trans}_{\langle q,a,q'a',\mathsf{left}\rangle}(x) \leftarrow \quad \mathsf{pushright}(a'(b(q'(x)))), \mathsf{check}_{n-1,b}(x), \mathsf{check}_{n-2,q}(x), \mathsf{check}_{n-3,a}(x)$$

for all $b \in \Sigma$.

The predicate $\mathsf{pushright}$ holds for words of the form $uw\#w'$ such that w is a configuration whose successor $uw\#w'w''$ leads to acceptance, where $w'' \in \Sigma^*$. Operationally, this predicate is responsible for guessing a correct suffix of the successor configuration. It is defined by

$$\mathsf{pushright}(x) \leftarrow \mathsf{pushright}(a(x)), \mathsf{check}_{n-1,a}(x)$$

for all $a \in \Sigma$ and

$$\mathsf{pushright}(x) \leftarrow \mathsf{leads\_to\_acc}(\#(x)), \mathsf{check}_{n-1,\#}(x).$$

This ends the construction of the program. We have the following theorem (remember that we identify strings with their term representation).

Theorem 1 *The atom* $\mathsf{leads\_to\_acc}(q_0 w \flat^{P(|w|)-|w|}\#)$ *belongs to the least model of the program above if and only if the machine M accepts the word w.*

Proof. It is enough to show that there exists a proof tree for the atom $\mathsf{leads\_to\_acc}(q_0 w \flat^{P(|w|)-|w|}\#)$ and the program above if and only if there exists an accepting computation tree for the machine M and the word w.

Recall that $n = P(|w|) + 2$, so the word $q_0 w \flat^{P(|w|)-|w|}\#$ has the length exactly n.

We start with the 'if' direction. Suppose that M accepts w. Take an accepting computation tree T_C for M and w. We will inductively build a proof tree T_P, such that for every node labeled $\mathsf{leads\_to\_acc}(p)$ in T_P there will be a corresponding node labeled c in T_C, where p is a path leading to c in T_C. We start from the node $\mathsf{leads\_to\_acc}(q_0 w \flat^{P(|w|)-|w|}\#)$.

Suppose that we have constructed a partial proof tree and take a node labeled $\mathsf{leads\_to\_acc}(p\#)$ whose children are not yet defined. If the last configuration c_p of the path p is final, we simply choose the clause $\mathsf{leads\_to\_acc}(\#(x)) \leftarrow \mathsf{final}(x)$ and follow the construction of the proof tree for the call $\mathsf{final}(p)$. If c_p is existential, then let c be the configuration following c_p in T_C, and let $\delta = \langle q, a, q', a', m\rangle$ be the transition leading from c_p to c. By choosing the clause $\mathsf{leads\_to\_acc}(\#(x)) \leftarrow \mathsf{reads}_{q,a}(x), \mathsf{succ}_{\langle q,a,q',a',m\rangle}(\#(x))$ we are sure that the goal $\mathsf{reads}_{q,a}(x)$ succeeds, and that the goal $\mathsf{succ}_{\langle q,a,q',a',m\rangle}(\#(x))$, after following the nondeterministic choices that correspond to the consecutive symbols in the configuration of c (so that all the check-predicates succeed), yields a goal $\mathsf{leads\_to\_acc}(p\#c\#)$ that corresponds to the node labeled c in T_C. If the configuration c_p is universal, then by following the clause

$$\mathsf{leads\_to\_acc}(\#(x)) \quad \leftarrow \quad \mathsf{reads}_{q,a}(x), \bigwedge_{\langle q',a',m\rangle \in \Delta(q,a)} \mathsf{succ}_{\langle q,a,q',a',m\rangle}(\#(x))$$

and analogous construction we get an extension of the tree to the nodes corresponding to all successors of c_p in T_C. Since T_C is finite, our construction terminates giving a proof tree for the atom $\mathsf{leads\_to\_acc}(q_0 w \flat^{P(|w|)-|w|}\#)$, which shows that it belongs to the least model of the program.

Now we prove the 'only if' direction. Suppose that the goal $\mathsf{leads\_to\_acc}(q_0 w \flat^{P(|w|)-|w|}\#)$ succeeds and take a proof tree T_P for this atom. Consider two consecutive calls $\mathsf{leads\_to\_acc}(p\#c_1\#)$ and $\mathsf{leads\_to\_acc}(p\#c_1\#c_2\#)$ in this tree. Since the only way from a call to $\mathsf{leads\_to\_acc}$ to another call to $\mathsf{leads\_to\_acc}$ goes via calls to $\mathsf{pushleft}$ (which adds only symbols from Σ to the argument), one call to trans_δ (which adds one symbol from Q) and then via calls to $\mathsf{pushright}$ (which adds only symbols from Σ), the word c_2 belongs to the set $\Sigma^* Q \Sigma^*$. Since the call $\mathsf{check}_{n-1,\#}(p\#c_1\#c_2)$ succeeds, the word c_2 has the length exactly $n-1$ and thus is an encoding of a configuration of M of the same length as all other configurations that occur in T_P. Since all calls to check-predicates from the predicates $\mathsf{pushleft}_\delta$, trans_δ and $\mathsf{pushright}$ succeed, c_2 is a successor configuration of c_1, with transition δ. Hence, by removing from T_P all nodes that are not labeled with the predicate leads_to_acc and by replacing each label leads_to_acc($p\#c\#$) by the label c we obtain a computation tree for M and $w\flat^{P(|w|)-|w|}$. Now, removing unnecessary blank symbols gives a desired computation tree, which shows that M accepts w. $\square$

Corollary 2 *The type-checking problem for path-based approximation is DEXPTIME-complete.*

Corollary 3 *The satisfiability problem for unary definite set constraints is DEXPTIME-complete.*

Corollary 4 *The membership and emptiness problems for finite automata with alternation and push operations are DEXPTIME-complete.*

Corollary 5 *The reachability problem for alternating pushdown systems DEXPTIME-complete. Moreover, it requires time that is at least exponential in the size of the system.*

The lower bounds for the corollaries above follow directly from Theorem 1 and reductions presented at the end of Section 2. For Corollary 4 one only has to note that the emptiness problem is at least as hard as the membership problem. The upper bounds are proved in [11, 9, 6] (Corollaries 2 and 4), [7] (Corollary 3), and [3] (Corollaries 4 and 5).

4 Conclusion

We have proved that the type-checking problem for path-based analysis of logic programs is DEXPTIME-complete. This gives a negative answer to the long-standing question whether the complexity of set-based analysis can be improved by ignoring dependencies between arguments to all function symbols. It remains to be seen if more efficient implementations based on this restriction are possible for practical applications, but at least we know that there is no hope for asymptotically faster algorithms.

The result has also consequences in set constraints, automata theory and model checking. The perhaps most surprising is the consequence in automata theory: we presented probably the first natural problems concerning finite automata on words that are DEXPTIME-complete.

Appendix

As we already mentioned, path-based program analysis is often confused with the *path-closed* one. The DEXPTIME-hardness of the type-checking problem for path-closed approximation is known by the reduction of the emptiness problem for intersection of path-closed languages [21], but it is not known whether the problem is decidable at all.

Below we recall the basic idea of the path-closed analysis and show a small example exhibiting the difference between the three (set-based, path-closed and path-based) analyses. A further discussion on set-based and path-closed analyses can be found e.g. in [8].

A regular set of terms is called path-closed if it is recognizable by a top-down deterministic tree automaton. This is equivalent to other notions occurring in the literature: path-closed sets are also called tuple-distributive, discriminative or deterministic. The path-closure operator PC assigns to a given set S of ground terms the least path-closed set that contain S. For example, $PC(\{f(a,a), f(b,b)\}) = \{f(a,a), f(b,b), f(a,b), f(b,a)\}$.

The path-closed approximation of a logic program $\mathcal{P}$ is the least fixed point of the operator $PC \circ T_{\mathcal{P}}$ where $T_{\mathcal{P}}$ is the immediate consequence operator for the program $\mathcal{P}$. It is not known whether this least fixed point is a decidable set, it is also not known whether it is always a regular set.

Consider the following extension of the example from Introduction.

$$\begin{array}{l} p_1(f(a,b)) \\ p_2(f(a,a)) \\ p_3(f(b,b)) \\ q(x) \leftarrow p_1(x), p_2(x) \\ q(x) \leftarrow p_1(x), p_3(x) \\ r(x) \leftarrow p_1(x) \\ r(x) \leftarrow p_2(x) \\ r(x) \leftarrow p_3(x) \end{array}$$

The table below shows the sets assigned to the predicates q and r by the three analyses.

	q	r
set-based:	$\emptyset$	$\{f(a,b), f(a,a), f(b,b)\}$
path-closed:	$\emptyset$	$\{f(a,b), f(a,a), f(b,b), f(b,a)\}$
path-based:	$\{f(a,b)\}$	$\{f(a,b), f(a,a), f(b,b), f(b,a)\}$

References

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

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

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR'97*, volume 1243 of *LNCS*, pages 135–150, 1997.

[4] J. R. Büchi. Regular canonical systems. *Archiv Mathematische Logik und Grundlagenforschung*, 6:91–111, 1964. Reprint in Saunders Mac Lane, Dirk Siefkes, editors, The collected works of J. Richard Büchi, Springer-Verlag, 1990.

[5] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, Jan. 1981.

[6] W. Charatonik, D. McAllester, D. Niwiński, A. Podelski, and I. Walukiewicz. The Horn mu-calculus. In V. Pratt, editor, *Proceedings of the 13th IEEE Annual Symposium on Logic in Computer Science (LICS)*, 1998.

[7] 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.

[8] W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the Fifth International Static Analysis Symposium (SAS)*, volume 1503 of *LNCS*, pages 278–294, Pisa, Italy, 1998. Springer-Verlag.

[9] 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.

[10] T. Frühwirth. Type Inference by Program Transformation and Partial Evaluation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*. MIT Press, 1989.

[11] 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.

[12] N. Heintze. *Set based program analysis.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[13] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.

[14] 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.

[15] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. Technical Report CMU-CS-91-110, School of Computer Science, Carnegie Mellon University, Feb. 1991. 42 pages.

[16] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 281–298. Springer-Verlag, 1994.

[17] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[18] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984.

[19] 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 *LNCS*. Springer-Verlag, 1997.

[20] J. C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.

[21] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52:57–60, 1994.

[22] M. Vardi. An automata-theoretic approach to unary set constraints. unpublished manuscript.

[23] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10:125–153, 1991.