Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 212 (2008) 191–206
www.elsevier.com/locate/entcs

Checking Emptiness of Non-Deterministic Regular Types with Set Operators

Lunjin Lu

Oakland University

Abstract

An algorithm to decide the emptiness of a regular type expression with set operators given a set of parameterized type definitions is presented. The algorithm generalizes previous work in that tuple distributivity is not assumed and set operators are permitted in type expressions.

Keywords: Regular types, Emptiness, Subtyping, Tabulation

1 Introduction

Types play an important role in programming languages. They make programs easier to understand and help detect errors. A type checker [23,28] requires the programmer to declare types for components of a program and verifies at compile-time that the execution of the program is prosecuted without violating the declared types. A type inference system [12,15,6,18,17,20] infers types for program components from its usage in the program. In between a type reconstruction system would allow the programmer to declares types for some program components and infer types for others [21]. One of important issues in a type system is to decide if one type is a subtype of another. There are two approaches to solving the subtyping problem. The syntactic approach consists in defining the subtyping relation by axiomatizing it by a set of inductive or co-inductive rules [5]. The semantic approach interprets types as subsets of the domain of computation and defines the subtyping relation as the inclusion of denoted sets [11].

A *type* is a finite expression that denotes a possibly infinite set of ground terms. An integral part of any type system is its type language that specifies which sets of ground terms are types. Regular term languages [7], called regular types, have been used widely used as types [25,22,8,24,12,28,15,13,6,18,17,20]. Most type systems use *tuple distributive types* [22] which are strictly less powerful than regular types. Tuple distributive types are regular types closed under tuple distributive closure.

doi:10.1016/j.entcs.2008.04.062

Intuitively, the tuple distributive closure of a set of terms is the set of all terms constructed recursively by permuting each argument position among all terms that have the same function symbol [28]. Tuple distributive types are not closed under set union and complement.

This paper gives an algorithm to decide if a type expression denotes the empty set of terms. The subtyping problem can be reduced to the emptiness problem in our setting since our language of types is closed under set union, set intersection and set complement. The correctness of the algorithm is proved and its complexity is analyzed. The algorithm works on regular types with parametric type definitions [24]. We allow parametric and overloading polymorphism in type definitions. These types are useful both in compilers and other program manipulation tools such as debuggers because they are easy to understand for programmers. Type expressions may contain set operators with their usual interpretations. Set operators allow concise and intuitive representation of regular types.

Though using regular term languages as types allow us to make use of theoretical results in the field of tree automata [12], algorithms for testing the emptiness of tree automata cannot be applied directly as type definitions may be parameterized. For instance, in order to decide the emptiness of a type expression given a set of type definitions, it would be necessary to construct a tree automaton from the type expression and the set of type definitions before an algorithm for determining the emptiness of an tree automaton can be used. When type definitions are parameterized, this would make it necessary to construct a different automaton each time the emptiness of a type expression is tested. Thus, an algorithm that works directly with type definitions is desirable as it avoids this repeated construction of automata. This is the main contribution of this paper.

Attempts have been made in the past to find algorithms for regular types [22,12,28,27,9,8]. Dart and Zobel's work [9] is the only one to present decision algorithms for emptiness and inclusion problems for regular types without the tuple distributive restriction. Unfortunately, their algorithm for the inclusion problem is incorrect for regular types in general. See [19] for a counterexample. Moreover, the type language of Dart and Zobel is less expressive than that presented in this paper since it doesn't allow set operators and parameterized type definitions.

Set constraint solving has also been used in type checking and type inference [3,2,16,10]. However, algorithms proposed for set constraint solving [3,4,2,1] are not applicable to the emptiness problem we considered in this paper as they don't take type definitions into account.

The remainder of this paper is organized as follows. Section 2 describes our language of type expressions and type definitions. Section 3 presents our algorithm for testing if a type expression denotes an empty set of terms. Section 4 addresses the correctness of the algorithm. Section 5 presents the complexity of the algorithm and section 6 concludes the paper. Proofs in an appendix can be omitted in the final version of the paper.

2 Type Language

Let Σ be a fixed ranked alphabet. Each symbol in Σ is called a function symbol and has a fixed arity. The arity of a symbol f is denoted as $arity(f)$. Assume that Σ contains at least one constant that is a function symbol of arity 0. Let $\mathcal{T}(\Phi)$ be the set of all terms over Φ. $\mathcal{T}(\Sigma)$ is the set of all possible values that a program variable can take. We shall use regular term languages over Σ as types.

A type is represented by a ground term constructed from another ranked alphabet Π and $\{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\}$, called type constructors. It is assumed that $(\Pi \cup \{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\}) \cap \Sigma = \emptyset$. Thus, a type expression is a term in $\mathcal{T}(\Pi \cup \{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\})$.

Types are defined by type rules. A type rule is a production rule of the form $c(\zeta_1, \cdots, \zeta_m) \rightarrow \tau$ where $c \in \Pi$, $\zeta_1, \cdots, \zeta_m$ are different type parameters and $\tau \in \mathcal{T}(\Sigma \cup \Pi \cup \Xi_m)$ where $\Xi_m = \{\zeta_1, \cdots, \zeta_m\}$. The condition that every type parameter in the righthand side of a type rule occurs in its lefthand side is often referred to as type preserving [26] and is shared by other formalisms for defining types such as tree automata [7], regular term grammars [9] and regular unary logic programs [28]. Note that overloading of function symbols is permitted as a function symbol can appear in the righthand sides of many type rules. We denote by Δ the set of all type rules and define $\Xi \stackrel{\text{def}}{=} \bigcup_{c \in \Pi} \Xi_{arity(c)}$. $\langle \Pi, \Sigma, \Delta \rangle$ is a restricted form of context-free term grammar.

Example 2.1 Let $\Sigma = \{0, s(), nil, cons(,)\}$ and $\Pi = \{Nat, Even, List()\}$. The following set of type rules defines natural numbers, even numbers and lists.

$$\Delta = \left\{ \begin{array}{l} Nat \rightarrow 0 \mid s(Nat), \quad Even \rightarrow 0 \mid s(s(Even)), \\ List(\zeta) \rightarrow nil \mid cons(\zeta, List(\zeta)) \end{array} \right\}$$

where, for instance, $Nat \rightarrow 0 \mid s(Nat)$ is an abbreviation of two rules $Nat \rightarrow 0$ and $Nat \rightarrow s(Nat)$.

We assume that Δ is simplified in that τ in each type rule $c(\zeta_1, \cdots, \zeta_m) \rightarrow \tau$ is of the form $f(\tau_1, \cdots, \tau_n)$ such that each τ_j, for $1 \leq j \leq n$, is either in Ξ_m or of the form $d(\zeta'_1, \cdots, \zeta'_k)$ and $\zeta'_1, \cdots, \zeta'_k \in \Xi_m$. There is no loss of generality to use a simplified set of type rules since every set of type rules can be simplified by introducing new type constructors and rewriting and adding type rules in the spirit of [9].

Example 2.2 The following is the simplified version of the set of type rules in example 2.1. $\Sigma = \{0, s(), nil, cons(,)\}$, $\Pi = \{Nat, Even, Odd, List()\}$ and

$$\Delta = \left\{ \begin{array}{l} Nat \rightarrow 0 \mid s(Nat),\ Even \rightarrow 0 \mid s(Odd), \\ Odd \rightarrow s(Even), \quad List(\zeta) \rightarrow nil \mid cons(\zeta, List(\zeta)) \end{array} \right\}$$

A type valuation ϕ is a mapping from Ξ to $\mathcal{T}(\Pi \cup \{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\})$. The instance $\phi(R)$ of a production rule R under ϕ results from substituting $\phi(\zeta)$ for each occurrence of type parameter ζ in R. For instance, $List(Nat \sqcap (\sim Even)) \rightarrow cons(Nat \sqcap (\sim Even), List(Nat \sqcap (\sim Even)))$ is the instance of $List(\zeta) \rightarrow cons(\zeta, List(\zeta))$ under $\phi = \{\zeta \mapsto Nat \sqcap (\sim Even)\}$. Let

$$ground(\Delta) \stackrel{\text{def}}{=} \{\phi(R) \mid R \in \Delta \wedge \phi \in (\Xi \mapsto \mathcal{T}(\Pi \cup \{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\}))\} \\ \cup \ \{\mathbf{1} \mapsto f(\mathbf{1}, \cdots, \mathbf{1}) \mid f \in \Sigma\}$$

$ground(\Delta)$ is the set of all ground instances of grammar rules in Δ plus rules of the form $\mathbf{1} \to f(\mathbf{1}, \cdots, \mathbf{1})$ for every $f \in \Sigma$.

Given a set Δ of type definitions, the type denoted by a type expression is determined by the following meaning function.

$$[\![\mathbf{1}]\!]_\Delta \stackrel{\text{def}}{=} \mathcal{T}(\Sigma), \qquad [\![\mathbf{0}]\!]_\Delta \stackrel{\text{def}}{=} \emptyset, \qquad [\![\sim E]\!]_\Delta \stackrel{\text{def}}{=} \mathcal{T}(\Sigma) - [\![E]\!]_\Delta,$$
$$[\![E_1 \sqcap E_2]\!]_\Delta \stackrel{\text{def}}{=} [\![E_1]\!]_\Delta \cap [\![E_2]\!]_\Delta, \qquad [\![E_1 \sqcup E_2]\!]_\Delta \stackrel{\text{def}}{=} [\![E_1]\!]_\Delta \cup [\![E_2]\!]_\Delta,$$
$$[\![\omega]\!]_\Delta \stackrel{\text{def}}{=} \bigcup_{(\omega \to f(E_1, \cdots, E_n)) \in ground(\Delta)} \{f(t_1, \cdots, t_n) \mid \forall 1 \le i \le n.\ t_i \in [\![E_i]\!]_\Delta\}$$

where ω is a type whose main constructor is in Π. Type constructors $\sqcap$, $\sqcup$ and $\sim$ are interpreted by $[\![\cdot]\!]_\Delta$ as set intersection, set union and set complement with respect to $\mathcal{T}(\Sigma)$. Type $\mathbf{1}$ denotes $\mathcal{T}(\Sigma)$ and type $\mathbf{0}$ the empty set.

Example 2.3 Let Δ be that in example 2.2. We have

$$[\![Nat]\!]_\Delta = \{0, s(0), s(s(0)), \cdots\}$$
$$[\![Even]\!]_\Delta = \{0, s(s(0)), s(s(s(s(0)))), \cdots\}$$
$$[\![Nat \sqcap \sim Even]\!]_\Delta = \{s(0), s(s(s(0))), s(s(s(s(s(0))))), \cdots\}$$
$$[\![List(Nat \sqcap \sim Even)]\!]_\Delta = \{cons(s(0), nil), cons(s(s(s(0))), nil), \cdots\}$$

Lemma 2.4 $[\![\mathcal{M}]\!]_\Delta$ *is a regular term language for any type expression* $\mathcal{M}$.

We extend $[\![\cdot]\!]_\Delta$ to sequences θ of type expressions: $[\![\epsilon]\!]_\Delta \stackrel{\text{def}}{=} \{\epsilon\}$ and $[\![\langle E \rangle \bullet \theta']\!]_\Delta \stackrel{\text{def}}{=} [\![E]\!]_\Delta \times [\![\theta']\!]_\Delta$ where ϵ is the empty sequence, $\bullet$ is the infix sequence concatenation operator, $\langle E \rangle$ is the sequence consisting of the type expression E and $\times$ is the Cartesian product operator. As a sequence of type expressions, ϵ can be thought of consisting of zero instance of $\mathbf{1}$. We use Λ to denote the sequence consisting of zero instance of $\mathbf{0}$ and define $[\![\Lambda]\!]_\Delta = \emptyset$.

We shall call a sequence of type expressions simply a sequence. A sequence expression is an expression consisting of sequences of the same length and $\sqcap$, $\sqcup$ and $\sim$. The length of the sequences in a sequence expression θ is called the dimension of θ and is denoted by $\|\theta\|$. Let θ, θ_1 and θ_2 be sequence expressions of the same length and

$$[\![\theta_1 \sqcap \theta_2]\!]_\Delta \stackrel{\text{def}}{=} [\![\theta_1]\!]_\Delta \cap [\![\theta_2]\!]_\Delta \qquad [\![\theta_1 \sqcup \theta_2]\!]_\Delta \stackrel{\text{def}}{=} [\![\theta_1]\!]_\Delta \cup [\![\theta_2]\!]_\Delta$$
$$[\![\sim\theta]\!]_\Delta \stackrel{\text{def}}{=} \underbrace{\mathcal{T}(\Sigma) \times \cdots \times \mathcal{T}(\Sigma)}_{\|\theta\| \text{ times}} - [\![\theta]\!]_\Delta$$

A conjunctive sequence expression is a sequence expression of the form $\gamma_1 \wedge \cdots \wedge \gamma_m$ where each γ_i, for $1 \le i \le m$, is a sequence.

3 Emptiness Algorithm

This section presents an algorithm that decides if a type expression denotes the empty set with respect to a given set of type definitions. The algorithm can also be used to decide if (the denotation of) one type expression is included in (the denotation of) another because E_1 is included in E_2 iff $E_1 \sqcap \sim E_2$ is empty.

We first introduce some terminology and notations. A type atom is a type expression of which the principal type constructor is not a set operator. A type literal is either a type atom or the complement of a type atom. A conjunctive type expression C is of the form $\sqcap_{i\in I} L_i$ with L_i being a type literal. Let α be a type atom. $\mathcal{F}(\alpha)$ defined below is the set of the principal function symbols of the terms in $[\![\alpha]\!]_\Delta$.

$$\mathcal{F}(\alpha) \stackrel{\text{def}}{=} \{f \in \Sigma \mid \exists \zeta_1 \cdots \zeta_k.((\alpha \to f(\zeta_1, \cdots, \zeta_k)) \in \mathit{ground}(\Delta))\}$$

Let $f \in \Sigma$. Define

$$\mathcal{A}^f_\alpha \stackrel{\text{def}}{=} \{\langle \alpha_1, \cdots, \alpha_k\rangle \mid (\alpha \to f(\alpha_1, \cdots, \alpha_k)) \in \mathit{ground}(\Delta)\}$$

We have $[\![\mathcal{A}^f_\alpha]\!]_\Delta = \{\langle t_1, \cdots, t_k\rangle \mid f(t_1, \cdots, t_k) \in [\![\alpha]\!]_\Delta\}$. Both $\mathcal{F}(\alpha)$ and $\mathcal{A}^f_\alpha$ are finite even though $\mathit{ground}(\Delta))$ is usually not finite.

The algorithm repeatedly reduces the emptiness problem of a type expression to the emptiness problems of sequence expressions and then reduces the emptiness problem of a sequence expression to the emptiness problems of type expressions. Tabulation is used to break down any possible loop and to ensure termination. Let O be a type expression or a sequence expression. Define $\text{empty}(O) \stackrel{\text{def}}{=} ([\![O]\!]_\Delta = \emptyset)$.

3.1 Two Reduction Rules

We shall first sketch the two reduction rules and then add tabulation to form an algorithm. Initially the algorithm is to decide the validity of a formula of the form

$$\text{empty}(E) \tag{1}$$

where E is a type expression.

3.1.1 Reduction Rule One.

The first reduction rule rewrites a formula of the form (1) into a conjunction of formulae of the following form.

$$\text{empty}(\sigma) \tag{2}$$

where σ is a sequence expression where $\sim$ is applied to type expressions but not to any sequence expression.

It is obvious that a type expression has a unique (modulo equivalence of denotation) disjunctive normal form. Let $\text{DNF}(E)$ be the disjunctive normal form of E. Then $\text{empty}(E)$ can written into $\wedge_{C\in\text{DNF}(E)}\text{empty}(C)$. Each C is a conjunctive type expression. We assume that C contains at least one positive type literal. This doesn't cause any loss of generality as $[\![\mathbf{1}\sqcap C]\!]_\Delta = [\![C]\!]_\Delta$ for any conjunctive type

expression C. We also assume that C doesn't contain repeated occurrences of the same type literal.

Let $C = \sqcap_{1\leq i\leq m}\omega_i \sqcap \sqcap_{1\leq j\leq n}\sim\tau_j$ where ω_i and τ_j are type atoms. The set of positive type literals in C is $pos(C) \stackrel{\text{def}}{=} \{\omega_i \mid 1 \leq i \leq m\}$, the set of complemented type atoms is $neg(C) \stackrel{\text{def}}{=} \{\tau_j \mid 1 \leq j \leq n\}$. Let $lit(C)$ denote the set of literals occurring in C. The following equivalence holds.

Lemma 3.1 empty(C) *is equivalent to*

$$\forall f \in (\cap_{\alpha\in pos(C)}\mathcal{F}(\alpha)).\ \text{empty}((\sqcap_{\omega\in pos(C)}(\sqcup\mathcal{A}^f_\omega))\sqcap(\sqcap_{\tau\in neg(C)}\sim(\sqcup\mathcal{A}^f_\tau))) \tag{3}$$

The intuition behind the equivalence is as follows. $[\![C]\!]_\Delta$ is empty iff, for every function symbol f, the set of the sequences $\langle t_1,\cdots,t_k\rangle$ of terms such that $f(t_1,\cdots,t_k) \in [\![C]\!]_\Delta$ is empty. Only the function symbols in $\cap_{\alpha\in pos(C)}\mathcal{F}(\alpha)$ need to be considered.

We note the following two special cases of the formula (3).

(a) If $\cap_{\alpha\in pos(C)}\mathcal{F}(\alpha) = \emptyset$ then the formula (3) is true because $\wedge\emptyset$ = true. In particular, $\mathcal{F}(\mathbf{0}) = \emptyset$. Thus, if $\mathbf{0} \in pos(C)$ then $\cap_{\alpha\in pos(C)}\mathcal{F}(\alpha) = \emptyset$ and hence the formula (3) is true.

(b) If $\mathcal{A}^f_\tau = \emptyset$ for some $\tau \in neg(C)$ then $\sqcup\mathcal{A}^f_\tau = \langle\mathbf{0},\cdots,\mathbf{0}\rangle$ and $\sim(\sqcup\mathcal{A}^f_\tau) = \langle\mathbf{1},\cdots,\mathbf{1}\rangle$. Thus, τ has no effect on the subformula for f when $\mathcal{A}^f_\tau = \emptyset$.

In order to get rid of complement operators over sequence sub-expressions, the complement operator in $\sim(\sqcup\mathcal{A}^f_\tau)$ is pushed inwards by the function *push* defined in the following.

$$push(\sim(\sqcup_{i\in I}\gamma_i)) \stackrel{\text{def}}{=} \sqcap_{i\in I} push(\sim\gamma_i)$$

$$push(\sim\langle E_1, E_2,\cdots,E_k\rangle) \stackrel{\text{def}}{=} \sqcup_{1\leq l\leq k}\langle \underbrace{\mathbf{1},\cdots,\mathbf{1}}_{l-1}, \sim E_l, \underbrace{\mathbf{1},\cdots,\mathbf{1}}_{k-l}\rangle \quad \text{for } k \geq 1$$

$$push(\sim\epsilon) \stackrel{\text{def}}{=} \Lambda$$

It follows from De Morgan's law and the definition of $[\![\cdot]\!]_\Delta$ that $[\![push(\sim(\sqcup\mathcal{A}^f_\tau))]\!]_\Delta = [\![\sim(\sqcup\mathcal{A}^f_\tau)]\!]_\Delta$. Substituting $push(\sim(\sqcup\mathcal{A}^f_\tau))$ for $\sim(\sqcup\mathcal{A}^f_\tau)$ in the formula (3) gives rise to a formula of the form (2).

3.1.2 Reduction Rule Two.

The second reduction rule rewrites a formula of the form (2) to a conjunction of disjunctions of formulae of the form (1). Formula (2) is written into a disjunction of formulae of the form.

$$\text{empty}(\Gamma)$$

where Γ be a conjunctive sequence expression. Let $\Gamma = \gamma_1\sqcap\cdots\sqcap\gamma_k$, $\Gamma{\downarrow}j \stackrel{\text{def}}{=} \sqcap_{1\leq i\leq k}\gamma^j_i$ with γ^j_i being the j^{th} component of γ_i.

In the case $\|\Gamma\| = 0$, empty(Γ) can be decided without further reduction. If $\Lambda \in \Gamma$ then empty(Γ) is true because $[\![\Lambda]\!]_\Delta = \emptyset$. Otherwise, empty$(\Gamma)$ is false because $[\![\Gamma]\!]_\Delta = \{\epsilon\}$.

In the case $\|\Gamma\| \neq 0$,

Lemma 3.2 $\mathrm{empty}(\Gamma)$ *is equivalent to* $\vee_{1\leq j\leq\|\Gamma\|}\mathrm{empty}(\Gamma{\downarrow}j)$.

Note that $\Gamma{\downarrow}j$ is a type expression and $\mathrm{empty}(\Gamma{\downarrow}j)$ is of the form (1).

3.2 Algorithm

The two reduction rules in the previous section form the core of the algorithm. However, they alone cannot be used as an algorithm because a formula $\mathrm{empty}(E)$ may reduce to a formula containing $\mathrm{empty}(E)$ as a sub-formula, leading to nontermination. Suppose $\Sigma = \{f(), a\}$, $\Pi = \{Null\}$ and $\Delta = \{Null \rightarrow f(Null)\}$. Clearly, $\mathrm{empty}(Null)$ is true. However, by the first reduction rule, $\mathrm{empty}(Null)$ reduces to $\mathrm{empty}(\langle Null\rangle)$ which then reduces to $\mathrm{empty}(Null)$ by the second reduction rule. This process will not terminate.

The solution, inspired by [9], is to remember in a table a particular kind of formulae of which truth is being tested. When a formula of that kind is tested, the table is first looked up. If the formula is implied by any formula in the table, then it is determined as true. Otherwise, the formula is added into the table and then reduced by a reduction rule.

The emptiness algorithm presented below remembers every conjunctive type expression of which emptiness is being tested. Thus the table is a set of conjunctive type expressions. Let C_1 and C_2 be conjunctive type expressions. We define $(C_1 \preceq C_2) \stackrel{\mathrm{def}}{=} (lit(C_1) \supseteq lit(C_2))$. Since $C_i = \sqcap_{L\in lit(C_i)}L$, $C_1 \preceq C_2$ implies $[\![C_1]\!]_\Delta \subseteq [\![C_2]\!]_\Delta$ and hence $(C_1 \preceq C_2) \wedge \mathrm{empty}(C_2)$ implies $\mathrm{empty}(C_1)$. Let $\mathcal{B}^f_C = (\sqcap_{\omega\in pos(C)}(\sqcup\mathcal{A}^f_\omega))\sqcap(\sqcap_{\tau\in neg(C)}push(\sim(\sqcup\mathcal{A}^f_\tau)))$. Adding tabulation to the two reduction rules, we obtain the following algorithm for checking emptiness of types.

$$(4)\qquad etype(E) \stackrel{\mathrm{def}}{=} etype(E, \emptyset)$$

$$(5)\qquad etype(E, \Psi) \stackrel{\mathrm{def}}{=} \forall C \in \mathrm{DNF}(E).etype\_conj(C, \Psi)$$

$$(6)\qquad etype\_conj(C, \Psi) \stackrel{\mathrm{def}}{=} \begin{cases} \text{true}, & \text{if } pos(C) \cap neg(C) \neq \emptyset, \\ \text{true}, & \text{if } \exists C' \in \Psi.C \preceq C', \\ \forall f \in \cap_{\alpha\in pos(C)}\mathcal{F}(\alpha).eseq(\mathcal{B}^f_C, \Psi \cup \{C\}), & \text{otherwise.} \end{cases}$$

$$(7)\qquad eseq(\Theta, \Psi) \stackrel{\mathrm{def}}{=} \forall\Gamma \in \mathrm{DNF}(\Theta).eseq\_conj(\Gamma, \Psi)$$

$$(8)\qquad eseq\_conj(\Gamma, \Psi) \stackrel{\mathrm{def}}{=} \begin{cases} \text{true} & \text{if } \|\Gamma\| = 0 \wedge \Lambda \in \Gamma, \\ \text{false} & \text{if } \|\Gamma\| = 0 \wedge \Lambda \notin \Gamma, \\ \exists 1 \leq j \leq \|\Gamma\|.etype(\Gamma{\downarrow}j, \Psi) & \text{if } \|\Gamma\| \neq 0. \end{cases}$$

Equation 4 initialises the table to the empty set. Equations 5 and 6 implement the first reduction rule while equations 7 and 8 implement the second reduction rule. $etype(,)$ and $etype\_conj(,)$ test the emptiness of an arbitrary type expression and that of a conjunctive type expression respectively. $eseq(,)$ tests emptiness of a sequence expression consisting of sequences and $\sqcap$ and $\sqcup$ operators while $eseq\_conj(,)$ tests the emptiness of a conjunctive sequence expression. The expression of which emptiness is to be tested is passed as the first argument to these functions. The table is passed as the second argument. It is used in $etype\_conj(,)$ to detect a conjunctive type expression of which emptiness is implied by the emptiness of a tabled

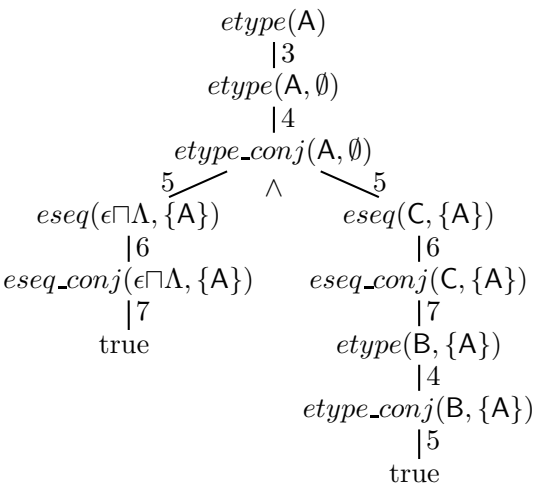


Fig. 1. Evaluation of $etype(Nat \sqcap \sim Even \sqcap \sim Odd)$)

conjunctive type expression. As we shall show later, this ensures the termination of the algorithm. Each of the four binary functions returns true iff the emptiness of the first argument is implied by the second argument and the set of type definitions.

Tabling any other kind of expressions such as arbitrary type expressions can also ensure termination. However, tabling conjunctive type expressions makes it easier to detect the implication of the emptiness of one expression by that of another because $lit(C)$ can be easily computed given a conjunctive type expression C which can be represented as $lit(C)$ in the table.

The first two definitions for $etype\_conj(C, \Psi)$ in equation 6 terminates the algorithm when the emptiness of C can be decided by C and Ψ without using type definitions. The first definition also excludes from the table any conjunctive type expression that contains both a type atom and its complement.

3.3 Examples

We now illustrate the algorithm with some examples.

Example 3.3 Let type definitions be given as in example 2.2. The tree in figure 1 depicts the evaluation of $etype(Nat \sqcap \sim Even \sqcap \sim Odd)$ by the algorithm. Nodes are labeled with function calls. We will identity a node with its label. Arcs from a node to its children are labeled with the number of the equation that is used to evaluate the node. Abbreviations used in the labels are defined in the legend to the right of the tree. Though $[\![\mathsf{A}]\!]_\Delta = [\![\mathsf{B}]\!]_\Delta$, A and B are syntactically different type expressions. The evaluation returns true, verifying $[\![Nat \sqcap \sim Even \sqcap \sim Odd]\!]_\Delta = \emptyset$. Consider $etype\_conj(\mathsf{B}, \{\mathsf{A}\})$. We have $\mathsf{B} \preceq \mathsf{A}$ as $lit(\mathsf{A}) = lit(\mathsf{B})$. Thus, by equation 6, $etype\_conj(\mathsf{B}, \{\mathsf{A}\}) = \text{true}$.

Example 3.4 Let type definitions be given as in example 2.2. The tree in figure 2 depicts the evaluation of $etype(List(Even \sqcap \sim Nat))$ by the algorithm. It returns false, verifying $[\![List(Even \sqcap \sim Nat)]\!]_\Delta \neq \emptyset$. Indeed, $[\![List(Even \sqcap \sim Nat)]\!]_\Delta = \{nil\}$.

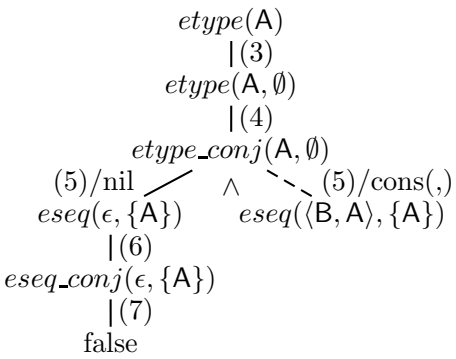


Fig. 2. Evaluation of $etype(List(Even \sqcap \sim Nat))$

The rightmost node is not evaluated as its sibling returns false, which is enough to establish the falsity of their parent node.

Example 3.5 The following is a simplified version of the type definitions that is used in [19] to show the incorrectness of the algorithm by Dart and Zobel for testing inclusion of one regular type in another [9].

Let $\Pi = \{\alpha, \beta, \theta, \sigma, \omega, \zeta, \eta\}$, $\Sigma = \{a, b, g(), h(,)\}$ and

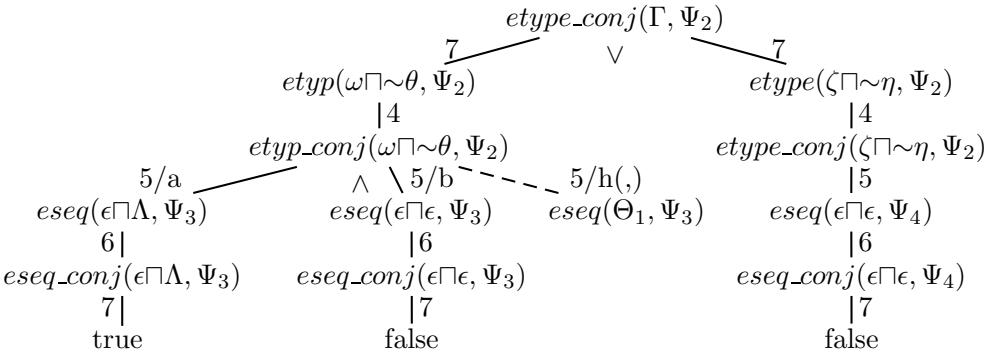
$$\Delta = \left\{ \begin{array}{l} \alpha \to g(\omega),\ \beta \to g(\theta) \mid g(\sigma),\ \ \theta \to a \mid h(\theta, \zeta),\ \sigma \to b \mid h(\sigma, \eta), \\ \omega \to a \mid b \mid h(\omega, \zeta) \mid h(\omega, \eta),\ \zeta \to a, \qquad\qquad \eta \to b \end{array} \right\}$$

Let $t = g(h(h(a, b), a))$. $t \in [\![\alpha]\!]_\Delta$ and $t \notin [\![\beta]\!]_\Delta$, see example 3 in [19] for more details. So, $[\![\alpha]\!]_\Delta \not\subseteq [\![\beta]\!]_\Delta$. This is verified by our algorithm as follows. Let $\Psi_1 = \{\alpha \sqcap \sim\beta\}$ and $\Psi_2 = \Psi_1 \cup \{\omega \sqcap \sim\theta \sqcap \sim\sigma\}$. By applying equations 4, 5, 6, 7, 8 and 5 in that order, we have $etype(\alpha \sqcap \sim\beta) = etype\_conj(\omega \sqcap \sim\theta \sqcap \sim\sigma, \Psi_1)$. By equation 6, we have $etype(\alpha \sqcap \sim\beta) = eseq(\epsilon \sqcap \Lambda \sqcap \epsilon, \Psi_2) \wedge eseq(\epsilon \sqcap \epsilon \sqcap \Lambda, \Psi_2) \wedge eseq(\Theta, \Psi_2)$ where $\Theta = (\langle \omega, \zeta \rangle \sqcup \langle \omega, \eta \rangle) \sqcap (\langle \sim\theta, \mathbf{1} \rangle \sqcup \langle \mathbf{1}, \sim\zeta \rangle) \sqcap (\langle \sim\sigma, \mathbf{1} \rangle \sqcup \langle \mathbf{1}, \sim\eta \rangle)$. We choose not to simplify expressions such as $\epsilon \sqcap \epsilon \sqcap \sim\Lambda$ so as to make the example easy to follow. By applying equations 7 and 8, we have both $eseq(\epsilon \sqcap \Lambda \sqcap \epsilon, \Psi_2) = \text{true}$ and $eseq(\epsilon \sqcap \epsilon \sqcap \Lambda, \Psi_2) = \text{true}$. So, $etype(\alpha \sqcap \sim\beta) = eseq(\Theta, \Psi_2)$. Let $\Gamma = \langle \omega, \zeta \rangle \sqcap \langle \sim\theta, \mathbf{1} \rangle \sqcap \langle \mathbf{1}, \sim\eta \rangle$. To show $etype(\alpha \sqcap \sim\beta) = \text{false}$, it suffices to show $eseq\_conj(\Gamma, \Psi_2) = \text{false}$ by equation 7 because $\Gamma \in \text{DNF}(\Theta)$ and $etype(\alpha \sqcap \sim\beta) = eseq(\Theta, \Psi_2)$.

Figure 3 depicts the evaluation of $eseq\_conj(\Gamma, \Psi_2)$. The node that is linked to its parent by a dashed line is not evaluated because one of its siblings returns false, which is sufficient to establish the falsity of its parent. It is clear from the figure that $etype\_conj(\Theta, \Psi_2) = \text{false}$ and hence $etype(\alpha \sqcap \sim\beta) = \text{false}$.

4 Correctness

This section addresses the correctness of the algorithm. We shall first show that tabulation ensures the termination of the algorithm because the table can only be of finite size. We then establish the partial correctness of the algorithm.



Legend:

$\Theta_1 = (\langle\omega, \zeta\rangle \sqcup \langle\omega, \eta\rangle) \sqcap (\langle\sim\theta, \mathbf{1}\rangle \sqcup \langle\mathbf{1}, \sim\zeta\rangle)$
$\Psi_3 = \Psi_2 \cup \{\omega \sqcap \sim\theta\}$
$\Psi_4 = \Psi_2 \cup \{\zeta \sqcap \sim\eta\}$
$\Gamma = \langle\omega, \zeta\rangle \sqcap \langle\sim\theta, \mathbf{1}\rangle \sqcap \langle\mathbf{1}, \sim\eta\rangle$

Fig. 3. Evaluation of $etype\_conj(\Gamma, \Psi_2)$

4.1 Termination

Given a type expression E, a top-level type atom in E is a type atom in E that is not a sub-term of any type atom in E. The set of top-level type atoms in E is denoted by $\mathrm{TLA}(E)$. For instance, letting $E = \sim List(Nat) \sqcup Tree(Nat \sqcap \sim Even)$, $\mathrm{TLA}(E) = \{List(Nat), Tree(Nat \sqcap \sim Even)\}$. We extend $\mathrm{TLA}(\cdot)$ to sequences by $\mathrm{TLA}(\langle E_1, E_2, \cdots, E_k\rangle) \stackrel{\mathrm{def}}{=} \bigcup_{1 \le i \le k} \mathrm{TLA}(E_i)$.

Given a type expression E_0, the evaluation tree for $etype(E_0)$ contains nodes of the form $etype(E, \Psi)$, $etype\_conj(C, \Psi)$, $eseq(\Theta, \Psi)$ and $eseq\_conj(\Gamma, \Psi)$ in addition to the root that is $etype(E_0)$. Only nodes of the form $etype\_conj(C, \Psi)$ add conjunctive type expressions to the table. Other forms of nodes only pass the table around. Therefore, it suffices to show that the type atoms occurring in the first argument of the nodes are from a finite set because any conjunctive type expression added into the table is the first argument of a node of the form $etype\_conj(C, \Psi)$.

The set $\mathrm{RTA}(E_0)$ of type atoms relevant to a type expression E_0 is the smallest set of type atoms satisfying

- $\mathrm{TLA}(E_0) \subseteq \mathrm{RTA}(E_0)$, and
- if τ is in $\mathrm{RTA}(E_0)$ and $\tau \to f(\tau_1, \tau_2, \cdots, \tau_k)$ is in $ground(\Delta)$ then $\mathrm{TLA}(\tau_i) \subseteq \mathrm{RTA}(E_0)$ for $1 \le i \le k$.

The height of τ_i is no more than that of τ for any $\tau \to f(\tau_1, \tau_2, \cdots, \tau_k)$ in $ground(\Delta)$. Thus, the height of any type atom in $\mathrm{RTA}(E_0)$ is finite. There are only a finite number of type constructors in Π. Thus, $\mathrm{RTA}(E_0)$ is of finite size. It follows by examining the algorithm that type atoms in the first argument of the nodes in the evaluation tree for $etype(E_0)$ are from $\mathrm{RTA}(E_0)$ which is finite. Therefore, the algorithm terminates.

4.2 Partial Correctness

The partial correctness of the algorithm is established by showing $etype(E_0) = \text{true}$ iff $\text{empty}(E_0)$. Let Ψ be a set of conjunctive type expressions. Define $\rho_\Psi \stackrel{\text{def}}{=} \wedge_{C\in\Psi}\text{empty}(C)$. The following two lemmas form the core of our proof of the partial correctness of the algorithm.

Lemma 4.1 *Let Ψ be a set of conjunctive type expressions, E a type expression, C a conjunctive type expression, Θ a sequence expression and Γ a conjunctive sequence expression.*

(a) If $\rho_\Psi \models \text{empty}(C)$ *then* $etype\_conj(C, \Psi) = \text{true}$, *and*

(b) If $\rho_\Psi \models \text{empty}(E)$ *then* $etype(E, \Psi) = \text{true}$, *and*

(c) If $\rho_\Psi \models \text{empty}(\Gamma)$ *then* $etype(\Gamma, \Psi) = \text{true}$, *and*

(d) If $\rho_\Psi \models \text{empty}(\Theta)$ *then* $etype(\Theta, \Psi) = \text{true}$.

Proof. The proof is done by induction on the size of the complement of Ψ with respect to the set of all possible conjunctive type expressions in which type atoms are from $\text{RTA}(E_0)$ where E_0 is a type expression.

Basis. The complement is empty. Ψ contains all possible conjunctive type expressions in which type atoms are from $\text{RTA}(E_0)$. We have $C \in \Psi$ and hence $etype\_conj(C, \Psi) = \text{true}$ by equation 6. Therefore, (a) holds. (b) follows from (a) and equation 5. (c) follows from (b), equation 8 and lemma 3.2, and (d) follows from (c) and equation 7.

Induction. By lemma 3.1 in the appendix, $\rho_\Psi \models \text{empty}(C)$ implies $\rho_\Psi \models \text{empty}(\mathcal{B}^f_C)$ for any $f \in \cap_{\alpha\in pos(C)}\mathcal{F}(\alpha)$. Thus, $\rho_{\Psi\cup\{C\}} \models \text{empty}(\mathcal{B}^f_C)$. The complement of $\Psi\cup\{C\}$ is smaller than the complement of Ψ. By the induction hypothesis, we have $eseq(\mathcal{B}^f_C, \Psi\cup\{C\}) = \text{true}$. By equation 6, $etype\_conj(C, \Psi) = \text{true}$. Therefore, (a) holds. (b) follows from (a) and equation 5. (c) follows from (b), equation 8 and lemma 3.2 and (d) follows from (c) and equation 7. □

Lemma 4.1 establishes the completeness of $etype(,)$, $etype\_conj(,)$, $eseq(,)$ and $eseq\_conj(,)$ while the following lemma establishes their soundness.

Lemma 4.2 *Let Ψ be a set of conjunctive type expressions, E a type expression, C a conjunctive type expression, Θ a sequence expression and Γ a conjunctive sequence expression.*

(a) $\rho_\Psi \models \text{empty}(C)$ *if* $etype\_conj(C, \Psi) = \text{true}$, *and*

(b) $\rho_\Psi \models \text{empty}(E)$ *if* $etype(E, \Psi) = \text{true}$, *and*

(c) $\rho_\Psi \models \text{empty}(\Gamma)$ *if* $etype(\Gamma, \Psi) = \text{true}$, *and*

(d) $\rho_\Psi \models \text{empty}(\Theta)$ *if* $etype(\Theta, \Psi) = \text{true}$.

Proof. It suffices to prove (a) since (b),(c) and (d) follow from (a) as in lemma 4.1. The proof is done by induction on $dp(C, \Psi)$ the depth of the evaluation tree for $etype\_conj(C, \Psi)$.

Basis. $dp(C,\Psi) = 1$. $etype\_conj(C,\Psi) = \text{true}$ implies either (i) $pos(C) \cap neg(C) \neq \emptyset$ or (ii) $\exists C' \in \Psi.C \preceq C'$. In case (i), empty$(C)$ is true and $\rho_\Psi \models$ empty(C). Consider case (ii). By the definition of $\preceq$ and ρ_Ψ, we have that $etype\_conj(C,\Psi) = \text{true}$ implies $\rho_\Psi \models$ empty(C).

Induction. $dp(C,\Psi) > 1$. Assume $etype\_conj(C,\Psi) = \text{true}$ and $\rho_\Psi \models \neg$empty(C). By lemma 3.1, there is $f \in \cap_{\alpha \in pos(C)} \mathcal{F}(\alpha)$ such that $\rho_\Psi \models \neg$empty$(\mathcal{B}_C^f)$. We have $\rho_{\Psi\cup\{C\}} \models \neg$empty$(\mathcal{B}_C^f)$. $dp(\mathcal{B}_C^f, \Psi \cup \{C\}) < dp(C,\Psi)$. By the induction hypothesis, we have $eseq(\mathcal{B}_C^f, \Psi \cup \{C\}) = \text{false}$ for otherwise, $\rho_{\Psi\cup\{C\}} \models \mathcal{B}_C^f$. By equation 6, $etype\_conj(C,\Psi) = \text{false}$ which contradicts $etype\_conj(C,\Psi) = \text{true}$. So, $\rho_\Psi \models$ empty(C) if $etype\_conj(C,\Psi) = \text{true}$. This completes the induction and the proof of the lemma. □

The following theorem is a corollary of lemmas 4.1 and 4.2.

Theorem 4.3 *For any type expression E, $etype(E) = \text{true}$ iff* empty(E).

Proof. By equation 4, $etype(E) = etype(E,\emptyset)$. By lemma 4.1.(b) and lemma 4.2.(b), we have $etype(E,\emptyset) = \text{true}$ iff $\rho_\emptyset \models$ empty(E). The result follows since $\rho_\emptyset = \text{true}$.□

5 Complexity

We now address the issue of complexity of the algorithm. We only consider the worst-case time complexity of the algorithm. The time spent on evaluating $etype(E_0)$ for a given type expression E_0 can be measured in terms of the number of nodes in the evaluation tree for $etype(E_0)$.

The algorithm cycles through $etype(,), etype\_conj(,), eseq(,)$ and $eseq\_conj(,)$. Thus, children of a node of the form $etype(E,\Psi)$ can only be of the form $etype\_conj(C,\Psi)$, and so on.

Let $|S|$ be the number of elements in a given set S. The largest possible table in the evaluation of $etype(E_0)$ contains all the conjunctive type expressions of which type atoms are from RTA(E_0). Therefore, the table can contain at most $2^{|\text{RTA}(E_0)|}$ conjunctive type expressions. So, the height of the tree is bounded by $\mathcal{O}(2^{|\text{RTA}(E_0)|})$.

We now show that the branching factor of the tree is also bounded by $\mathcal{O}(2^{|\text{RTA}(E_0)|})$. By equation 5, the number of children of $etype(E,\Psi)$ is bounded by two to the power of the number of type atoms in E which is bounded by $|\text{RTA}(E_0)|$ because E can only contain type atoms from RTA(E_0). By equation 6, the number of children of $etype\_conj(C,\Psi)$ is bounded by $|\Sigma|$. The largest number of children of a node $eseq(\Theta,\Psi)$ is bounded by two to the power of the number of sequences in Θ where $\Theta = \mathcal{B}_C^f$. For each $\tau \in neg(C)$, $|push(\sim(\sqcup \mathcal{A}_\tau^f))|$ is $\mathcal{O}(arity(f))$ and $|C| < |\text{RTA}(E_0)|$. Thus, the number of sequences in Θ is $\mathcal{O}(arity(f) * |\text{RTA}(E_0)|)$ and hence the number of children of $eseq(\Theta,\Psi)$ is $\mathcal{O}(2^{|\text{RTA}(E_0)|})$ since $arity(f)$ is a constant. By equation 8, the number of children of $eseq\_conj(\Gamma,\Psi)$ is bounded by $\max_{f\in\Sigma} arity(f)$. Therefore, the branching factor of the tree is bounded by $\mathcal{O}(2^{|\text{RTA}(E_0)|})$. The above discussion leads to the following conclusion.

Proposition 5.1 *The time complexity of the algorithm is $\mathcal{O}(2^{|\text{RTA}(E_0)|})$.*

The fact that the algorithm is exponential in time is expected because the complexity coincides with the complexity of deciding the emptiness of any tree automaton constructed from the type expression and the type definitions. A deterministic frontier-to-root tree automaton recognising $[E_0]_\Delta$ will consist of $2^{|\mathrm{RTA}(E_0)|}$ states as observed in the proof of lemma 2.4. It is well-known that the decision of the emptiness of the language of a deterministic frontier-to-root tree automaton takes time polynomial in the number of the states of the tree automaton. Therefore, the worst-case complexity of the algorithm is the best we can expect from an algorithm for deciding the emptiness of regular types that contain set operators.

6 Conclusion

We have presented an algorithm for deciding the emptiness of non-deterministic regular types with set operators. Type expressions are constructed from type constructors and set operators. Type definitions define the meaning of type expressions.

The algorithm uses tabulation to ensure termination. Though the tabulation is inspired by Dart and Zobel [9], the decision problem we consider in this paper is more complex as type expressions may contain set operators. For that reason, the algorithm can also be used for inclusion and equivalence problems of regular types. The way we use tabulation leads to a correct algorithm for regular types while the Dart-Zobel algorithm has been proved incorrect for regular types [19] in general.

In addition to correctness, our algorithm generalises the work of Dart and Zobel [9] in that type expressions can contain set operators and type definitions can be parameterised. Parameterised type definitions are more natural than monomorphic type definitions [12,23,28] while set operators makes type expressions concise. The combination of these two features allows more natural type declarations. For instance, the type of the logic program *append* can be declared or inferred as $append(List(\alpha), List(\beta), List(\alpha \sqcup \beta))$.

The algorithm is exponential in time. This coincides with deciding the emptiness of the language recognised by a tree automaton constructed from the type expression and the type definitions. However, the algorithm avoids the construction of the tree automaton which cannot be constructed *a priori* when type definitions are parameterised.

Another related field is set constraint solving [3,2,16,10]. However, set constraint solving methods are intended to infer regular tree languages as approximations to program properties rather than for checking the emptiness of regular types that are defined by a priori type definitions [24]. Therefore, they are useful in different settings from the algorithm presented in this paper. In addition, algorithms proposed for solving set constraints [3,4,2,1] are not applicable to the emptiness problem we considered in this paper. Take for example the *constructor rule* in [3,2] which states that emptiness of $f(E_1, E_2, \cdots, E_m)$ is equivalent to the emptiness of E_i for some $1 \leq i \leq m$. However, empty$(List(\mathbf{0}))$ is not equivalent to empty$(\mathbf{0})$. The latter is true while the former is false since $[List(\mathbf{0})]_\Delta = \{nil\}$. The constructor rule doesn't apply because it deals with function symbols only but doesn't take the type

definitions into account.

References

[1] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In *Proceedings of 1993 Computer Science Logic Conference*, pages 1–17, 1992

[2] A. Aiken and T.K. Lakshman. Directional type checking of logic programs. In B. Le Charlier, editor, *Proceedings of the First International Static Analysis Symposium*, pages 43–60. Springer-Verlag, 1994.

[3] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 329–340. The IEEE Computer Society Press, 1992.

[4] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[5] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

[6] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.

[7] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata, 2007.

[8] P.W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *The Journal of Logic Programming*, 14(1-2):31–69, 1992.

[9] P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–189. The MIT Press, 1992.

[10] P. Devienne, J-M. Talbot, and S. Tison. Co-definite set constraints with membership expressions. In J. Jaffar, editor, *Proceedings of the 1998 Joint Conference and Symposium on Logic Programming*, pages 25–39. The MIT Press, 1998.

[11] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proceedings of the Seventeenth IEEE Symposium on Logic in Computer Science*, pages 137–146. The IEEE Computer Society, 2002.

[12] T. Frühwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309. The IEEE Computer Society Press, 1991.

[13] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In M. Bruynooghe, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[14] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomma, editors, *Handbook of Formal Languages*, pages 1–68. Springer-Verlag, 1996.

[15] N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–155. The MIT Press, 1992.

[16] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In A. Borning, editor, *Principles and Practice of Constraint Programming*, pages 281–298. Springer, 1994.

[17] P. M. Hill and F. Spoto. Generalising *Def* and *Pos* to type analysis. *Journal of Logic and Computation*, 12(3):497–542, 2002.

[18] L. Lu. A polymorphic type analysis in logic programs by abstract interpretation. *The Journal of Logic Programming*, 36(1):1–54, 1998.

[19] L. Lu. On the Dart-Zobel algorithm for testing regular type inclusion. *SIGPLAN Notices*, 36(9):81–85, 2001.

[20] L. Lu. Improving precision of type analysis using non-discriminative union. *Theory and Practice of Logic Programming*, 8(1):33–79, 2008.

[21] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[22] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE international Symposium on Logic Programming*, pages 289–298. The IEEE Computer Society Press, 1984.

[23] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[24] U.S. Reddy. Types for logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 836–40. The MIT Press, 1990.

[25] M. Soloman. Type definitions with parameters. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 31–38, 1978.

[26] J. Tiuryn. Type inference problems: A survey. In B. Roven, editor, *Proceedings of the Fifteenth International Symposium on Mathematical Foundations of Computer Science*, pages 105–120. Springer-Verlag, 1990.

[27] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphically typed logic programs. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 379–93. The MIT Press, 1991.

[28] E. Yardeni and E. Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 10(2):125–153, 1991.

A Appendix

A.1 Proof of Lemma 3.1.

Let $\mathbf{t}$ be a sequence of terms and f a function symbol. By the definition of $[\![\cdot]\!]_\Delta$, $f(\mathbf{t}) \in [\![C]\!]_\Delta$ iff $f \in \cap_{\alpha\in pos(C)}\mathcal{F}(\alpha)$ and $\mathbf{t} \in [\![\sqcap_{\omega\in pos(C)}(\sqcup\mathcal{A}^f_\omega))]\!]_\Delta \setminus [\![(\sqcup_{\tau\in neg(C)}(\sqcup\mathcal{A}^f_\tau))]\!]_\Delta$.

$$\mathbf{t} \in [\![\sqcap_{\omega\in pos(C)}(\sqcup\mathcal{A}^f_\omega))]\!]_\Delta \setminus [\![(\sqcup_{\tau\in neg(C)}(\sqcup\mathcal{A}^f_\tau))]\!]_\Delta \quad \text{iff}$$
$$\mathbf{t} \in [\![(\sqcap_{\omega\in pos(C)}(\sqcup\mathcal{A}^f_\omega))\sqcap(\sqcap_{\tau\in neg(C)}\sim(\sqcup\mathcal{A}^f_\tau))]\!]_\Delta.$$

Thus, empty(C) holds iff empty$((\sqcap_{\omega\in pos(C)}(\sqcup\mathcal{A}^f_\omega))\sqcap(\sqcap_{\tau\in neg(C)}\sim(\sqcup\mathcal{A}^f_\tau)))$ holds for each $f \in \cap_{\alpha\in pos(C)}\mathcal{F}(\alpha)$. □

A.2 Proof of Lemma 3.2.

Let $\|\Gamma\| = n$ and $\Gamma = \gamma_1\sqcap\gamma_2\sqcap\cdots\sqcap\gamma_m$ with $\gamma_i = \langle\gamma_{i,1}, \gamma_{i,2}, \cdots, \gamma_{i,n}\rangle$. We have $[\![\Gamma]\!]_\Delta = \bigcap_{1\le j\le m}[\![\gamma_j]\!]_\Delta$. We have $\Gamma{\downarrow}j = \gamma_{1,j}\sqcap\gamma_{2,j}\sqcap\cdots\sqcap\gamma_{m,j}$. $\exists 1 \le j \le n.$empty$(\Gamma{\downarrow}j)$ iff $\exists 1 \le j \le n. \bigcap_{1\le i\le m}[\![\gamma_{i,j}]\!]_\Delta = \emptyset$ iff $[\![\Gamma]\!]_\Delta = \emptyset$ iff empty(Γ). □

A.3 Proof of Lemma 2.4.

The proof is done by constructing a regular term grammar for $\mathcal{M}$ [7]. We first consider the case $\mathcal{M} \in \mathcal{T}(\Pi \cup \{\mathbf{1}, \mathbf{0}\})$. Let $R = \langle \text{RTA}(\mathcal{M}), \Sigma, \emptyset, \Upsilon, \mathcal{M}\rangle$ with

$$\Upsilon = \{(\alpha \to f(\alpha_1, \cdots, \alpha_k)) \in ground(\Delta) \mid \alpha \in \text{RTA}(\mathcal{M})\}$$

R is a regular term grammar. It now suffices to prove that $t \in [\![\mathcal{M}]\!]_\Delta$ iff $\mathcal{M} \Rightarrow^*_R t$.

- Sufficiency. Assume $\mathcal{M} \Rightarrow^*_R t$. The proof is done by induction on derivation steps in $\mathcal{M} \Rightarrow^*_R t$.
  - · Basis. $\mathcal{M} \Rightarrow_R t$. t must be a constant and $\mathcal{M} \to t$ is in Υ which implies $\mathcal{M} \to t$ is in $ground(\Delta)$. By the definition of $[\![\cdot]\!]_\Delta$, $t \in [\![\mathcal{M}]\!]_\Delta$.
  - · Induction. Suppose $\mathcal{M} \Rightarrow f(\mathcal{M}_1, \cdots, \mathcal{M}_k) \Rightarrow^{(n-1)}_R t$. Then $t = f(t_1, \cdots, t_k)$ and $\mathcal{M}_i \Rightarrow^{n_i}_R t$ with $n_i \le (n-1)$. By the induction hypothesis, $t_i \in [\![\mathcal{M}_i]\!]_\Delta$ and hence $t \in [\![\mathcal{M}]\!]_\Delta$ by the definition of $[\![\cdot]\!]_\Delta$.

- Necessity. Assume $t \in [\![\mathcal{M}]\!]_\Delta$. The proof is done by the height of t, denoted as $height(t)$.
  - $height(t) = 0$ implies that t is a constant. $t \in [\![\mathcal{M}]\!]_\Delta$ implies that $\mathcal{M} \to t$ is in $ground(\Delta)$ and hence $\mathcal{M} \to t$ is in Υ. Therefore, $\mathcal{M} \Rightarrow_R t$.
  - Let $height(t) = n$. Then $t = f(t_1, \cdots, t_k)$. $t \in [\![\mathcal{M}]\!]_\Delta$ implies that $(\mathcal{M} \to f(\mathcal{M}_1, \cdots, \mathcal{M}_k)) \in ground(\Delta)$ and $t_i \in [\![\mathcal{M}_i]\!]_\Delta$. By the definition of Υ, we have $(\mathcal{M} \to f(\mathcal{M}_1, \cdots, \mathcal{M}_k)) \in \Upsilon$. By the definition of RTA$(\cdot)$, we have $\mathcal{M}_i \in \text{RTA}(\mathcal{M})$. By the induction hypothesis, $\mathcal{M}_i \Rightarrow_R^* t_i$. Therefore, $\mathcal{M} \Rightarrow_R f(\mathcal{M}_1, \cdots, \mathcal{M}_k) \Rightarrow_R^* f(t_1, \cdots, t_k) = t$.

Now consider the case $\mathcal{M} \in \mathcal{T}(\Pi \cup \{\sqcap, \sqcup, \sim, \mathbf{1}, \mathbf{0}\})$. We complete the proof by induction on the height of $\mathcal{M}$.

- $height(\mathcal{M}) = 0$. Then $\mathcal{M}$ doesn't contain set operator. We have already proved that $[\![\mathcal{M}]\!]_\Delta$ is a regular term language.
- Now suppose $height(\mathcal{M}) = n$. If $\mathcal{M}$ doesn't contain set operator then the lemma has already been proved. If the principal type constructor is one of set operators then the result follows immediately as regular term languages are closed under union, intersection and complement operators [14,7]. It now suffices to prove the case $\mathcal{M} = c(\mathcal{M}_1, \cdots, \mathcal{M}_\ell)$ with $c \in \Pi$. Let $\mathcal{N} = c(X_1, \cdots, X_\ell)$ where each X_j is a different new type constructor of arity 0.

  Let $\Pi' = \Pi \cup \{X_1, \cdots, X_\ell\}, \Sigma' = \Sigma \cup \{x_1, \cdots, x_\ell\}$ and $\Delta' = \Delta \cup \{X_j \to x_j | 1 \leq j \leq \ell\}$. $[\![\mathcal{N}]\!]_{\Delta'}$ is a regular term language on $\Sigma \cup \{x_1, \cdots, x_\ell\}$ because $\mathcal{N}$ doesn't contain set operators. By the induction hypothesis, $[\![\mathcal{M}_j]\!]_\Delta$ is a regular term language. By the definition of $[\![\cdot]\!]$, we have

$$[\![\mathcal{M}]\!]_\Delta = [\![\mathcal{N}]\!]_{\Delta'}[x_1 := [\![\mathcal{M}_1]\!]_\Delta, \cdots, x_\ell := [\![\mathcal{M}_\ell]\!]_\Delta]$$

  which is a regular term language [14,7]. $S[y_1 := S_{y_1}, \cdots,]$ is the set of terms each of which is obtained from a term in S by replacing each occurrence of y_j with a (possibly different) term from S_{y_j}. This completes the induction and the proof.

The proof also indicates that a non-deterministic frontier-to-root tree automaton that recognizes $[\![\mathcal{M}]\!]_\Delta$ has $|\text{RTA}(\mathcal{M})|$ states and that a deterministic frontier-to-root tree automaton that recognizes $[\![\mathcal{M}]\!]_\Delta$ has $\mathcal{O}(2^{|\text{RTA}(\mathcal{M})|})$ states. □