Solving Systems of Set Constraints (Extended Abstract)

Alexander Aiken Edward L. Wimmers

IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
phone: 408/927-1876 or 927-1882
email: *lastname*@almaden.ibm.com
fax: 408/927-2100

Abstract

Systems of set constraints are a natural formalism for many problems in program analysis. Set constraints are also a generalization of tree automata. We present an algorithm for solving systems of set constraints built from free variables, constructors, and the set operations of intersection, union, and complement. Furthermore, we show that all solutions of such systems can be finitely represented.[1]

1 Introduction

Set constraints are a natural formalism for describing relationships between sets of terms of a free algebra. A set constraint has the form $X \subseteq Y$, where X and Y are *set expressions*. Examples of set expressions are 0 (the empty set), 1 (the set of all terms), α (a set-valued variable), $c(X, Y)$ (a constructor application), and the union, intersection, or complement of set expressions. Given a set of constructors C where each $c \in C$ has arity $a(c)$, set expressions are defined by the grammar:

$$E ::= 0 \mid 1 \mid \alpha \mid c(E_1, \ldots, E_{a(c)}) \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E_1$$

Many computational problems requiring the solution of systems of set constraints arise in program analysis. In program analysis algorithms, sets of terms describe the possible values computed by a program. Set constraints are generated from the program text; solving the constraints yields some useful information about the program (e.g., for type-checking or optimization). For example, consider a program with two uses of a variable v. Assume that from one use we determine that v must be a number (i.e., $v \subseteq \text{Int} \cup \text{Float}$) and from the other use we determine that v cannot be an integer (i.e., $v \subseteq \neg\text{Int}$). Combining these constraints, we can infer that v must be a floating-point number (i.e., $v \subseteq (\text{Int} \cup \text{Float}) \cap \neg\text{Int} = \text{Float}$).

Set constraints have been used in program analysis and type inference algorithms for functional languages [1, 2, 10, 12, 14, 17][2], logic programming languages [8, 11], and imperative languages [9]. Solving a system of set constraints is central to each of these program analysis algorithms.

Systems of set constraints can also define tree automata; in this case, the sets of terms are the language of a finite-state machine. For a tree automaton A with states $\{a_1, a_2, \ldots\}$, a set S of constraints can be constructed over variables $\{\alpha_1, \alpha_2, \ldots\}$ such that the unique solution of the constraints assigns to α_i the set of terms accepted by state a_i.

Set constraints are perhaps the simplest formalism for expressing constraints between sets of terms. Despite this mathematical simplicity, the applications to program analysis, and their close relationship to tree automata, set constraints are relatively poorly understood. In this paper we present an algorithm for solving systems of set constraints; this algorithm produces a finite representation of all the solutions of a system

[1] To appear in the Proceedings of the 1992 IEEE Symposium on Logic in Computer Science

[2] In [17], an *ad hoc* formalism equivalent to set constraints is used.

of set constraints. We show that the complexity of deciding whether or not a system of set constraints is consistent (i.e., has a solution) is in NEXPTIME and is EXPTIME-hard. The exact complexity of the problem remains open.

Our primary contribution is showing that systems of set constraints that use all the standard set operations, especially unrestricted union and complement, can be solved. Set complement has received little attention in previous work on set constraints, but set complement (or something like it) is required to express the solutions of many set constraints, even those that do not explicitly use complement. For example, the solutions of the constraint $\alpha \cap \beta \subseteq 0$ are all substitutions in which α and β are disjoint—i.e., in which $\alpha \subseteq \neg\beta$.

The centerpiece of our development is an algorithm that incrementally transforms a system of constraints while preserving the set of solutions. Eventually, either the system is shown to be inconsistent or all solutions can be exhibited. Most of the work is in proving that if this algorithm does not discover an inconsistency, then the system has a solution. This is done by showing that the system of constraints generated by the algorithm can be transformed into an equivalent set of equations that are guaranteed to have a solution. These equations are essentially tree automata.

The rest of the paper is organized as follows. Section 2 contains basic definitions. Section 3 gives some examples illustrating the difficulty of solving set constraints; Section 4 covers related work. Section 5 gives an outline of the algorithm and sketches its proof of correctness. Sections 6 and 7 present the algorithm, while Sections 8 and 9 prove its correctness and show how to characterize the solutions of systems of set constraints. Section 10 discusses the complexity of the algorithm. Proofs of two lemmas are included in the appendix. The proof of Theorem 8.2 is presented in the body of the paper because it is the key step in the overall construction.

2 Definitions

Let $C = \{b, c, \ldots\}$ be a finite[3] set of constructors and let $V = \{\alpha, \beta, \ldots\}$ be a set of variables. Every constructor c has arity $a(c) \geq 0$. A *system of set of constraints* has the form $\{\ldots, X \subseteq Y, \ldots\}$, where X and Y are set expressions. We write $X = Y$ for the pair of constraints $X \subseteq Y$ and $Y \subseteq X$.

Definition 2.1 (Terms) Let $H^0 = \emptyset$ and let $H^{i+1} = H^i \cup \{c(t_1, \ldots, t_{a(c)}) | t_i \in H^i, c \in C\}$. The Herbrand Universe H is the least upper bound of the series $H^0 \subseteq H^1 \subseteq \ldots$.

We use H^i in set expressions as a syntactic abbreviation for the set of all terms in H^i. We assume that the Herbrand Universe is non-empty (i.e., there is at least one zero-ary constructor in C). A *substitution* σ is a function $\sigma : V \to \mathcal{P}(H)$ from variables to sets of terms. The standard semantics of set expressions maps an expression and a substitution to a set of terms:

$$\begin{aligned}
\mu(0, \sigma) &= \emptyset \\
\mu(1, \sigma) &= H \\
\mu(\alpha, \sigma) &= \sigma(\alpha) \\
\mu(\neg X, \sigma) &= H - \mu(X, \sigma) \\
\mu(X \cup Y, \sigma) &= \mu(X, \sigma) \cup \mu(Y, \sigma) \\
\mu(X \cap Y, \sigma) &= \mu(X, \sigma) \cap \mu(Y, \sigma) \\
\mu(c(X_1, \ldots, X_{a(c)}), \sigma) &= \\
\{c(t_1, \ldots, t_{a(c)}) | t_i \in \mu(X_i, \sigma),\ c \in C\}
\end{aligned}$$

Definition 2.2 (Solutions) The set of solutions $\mathcal{S}(S)$ of a system S of constraints is the set of all substitutions that satisfy the constraints: $\mathcal{S}(S) = \{\sigma | \mu(X, \sigma) \subseteq \mu(Y, \sigma)$ where $X \subseteq Y \in S\}$. If a system of constraints S is inconsistent, then $\mathcal{S}(S) = \emptyset$.

Definition 2.3 (Restricted Solutions) The solutions of a system S restricted to a set of variables is the set of substitutions in $\mathcal{S}(S)$ restricted to those variables: $\mathcal{S}(S) \uparrow \{\alpha_1, \ldots, \alpha_n\} = \{\sigma \uparrow \{\alpha_1, \ldots, \alpha_n\} | \sigma \in \mathcal{S}(S)\}$. We write $\mathcal{S}(S_1) =_V \mathcal{S}(S_2)$ for $\mathcal{S}(S_1) \uparrow V = \mathcal{S}(S_2) \uparrow V$.

3 Examples of Set Constraints

This section illustrates some of the difficulties in solving set constraints. Consider an equation $\alpha = E(\alpha)$. If $E()$ does not contain the negation symbol, then this equation has a solution; this follows immediately from the fact that all set expression operations except complement are monotonic. Presumably,

[3] The results are easily extended to infinite sets of constructors.

this is why equations of this form ($\alpha = E(\alpha)$ with no negations) have been studied almost exclusively [1, 2, 8, 12]. However, when set complement is added to the language, even equations of this restricted form might not have solutions. For example, the equation $\alpha = \neg\alpha$ has no solutions.

Another way to generalize constraints of the form $\alpha = E(\alpha)$ is to drop the requirement that the left-hand side of the equation be a single variable. However, if the constraint language permits arbitrary equations between set expressions (as our language does), then the constraint language already has all the power of complement and $\subseteq$. The constraint $E_1 \subseteq E_2$ can be expressed as $E_1 = E_1 \cap E_2$, and the expression $\neg E$ can be replaced by a fresh variable α with the added constraints $\alpha \cap E = 0$ and $\alpha \cup E = 1$.

Another problem is that set constraints do not necessarily have a least solution even when a solution exists. Consider the system $\{\alpha \cap \beta \subseteq 0, b \subseteq \alpha \cup \beta\}$. This system has two incomparable solutions: $\alpha = \{b\}$ and $\beta = 0$, or $\alpha = 0$ and $\beta = \{b\}$.

4 Related Work

Algorithms are known for a number of special cases of the general problem of solving a system of set constraints. These algorithms were developed either for an application in program analysis [7, 10, 12, 14] or in work on finite automata [3, 6, 13].

Mishra and Reddy [12] give a method for solving constraints between expressions without complement where all set unions are *discriminative* (i.e., if $c(\ldots) \cup d(\ldots)$ is an expression, then $c \neq d$). Heintze and Jaffar [7] give an algorithm for the class of *definite* constraints, which are of the form $A \subseteq B$, where B contains no set operations and A contains no complement operations. Heintze and Jaffar also use *projection functions*, an additional set operation we do not consider. For every constructor f, a family of projection functions $f^{-1}, \ldots, f^{-a(f)}$ is defined as follows:

$$\mu(f^{-i}(E), \sigma) = \{x_i | f(x_1, \ldots, x_{a(f)}) \in \mu(E, \sigma)\}$$

Reynolds [14] and Jones and Muchnick [10] present algorithms for solving set constraints with projection functions, but without union or complement operations.

In work on finite automata, Brzozowski and Leiss present an algorithm for solving equations between regular languages with free variables [3]. Since regular languages are equivalent to tree languages with only unary constructors, our algorithm can be regarded as the generalization of their work to constructors of arbitrary arity.

A related line of work is program analysis methods based on extensions of tree automata other than set constraints [4, 5]. The expressive power of these techniques is different than that of set constraints, but the exact relationship is as yet undetermined. We leave such comparisons as future work.

5 Outline of the Development

We give an algorithm that takes a system of constraints S and either reports that the constraints are inconsistent or produces a finite set of *solved form systems*. Solved form systems are sets of equations; these equations always have solutions and all solutions can be characterized easily.

To motivate the definition of solved form systems, recall that an equation of the form $\alpha = E$ has a solution whenever E has no negation symbols. Thus, eliminating negations would appear to be a positive step toward discovering the solutions of a system. Unfortunately, it is not always possible to eliminate negations in equations (e.g., consider the equation $\alpha = \neg\beta$), so the definition of solved form uses a weaker condition that merely restricts where negations can appear.

Another problem is that the solutions of an equation can be difficult to characterize, even if the equation has no negations. For example, consider the equation $\alpha = \alpha \cap \beta$. The solutions of this equation are $\{\sigma | \sigma(\alpha) \subseteq \sigma(\beta)\}$, but that fact is not obvious from the form of the equation. By restricting where the right-hand side variable α appears in an equation $\alpha = E$, we obtain equations whose solutions are characterized easily. The definition of solved form requires the following two technical definitions.

Definition 5.1 The *top-level variables* of a set expression are:

$$\begin{aligned} TLV(0) &= \emptyset \\ TLV(1) &= \emptyset \\ TLV(\alpha) &= \{\alpha\} \end{aligned}$$

$$
\begin{aligned}
TLV(\neg X) &= TLV(X) \\
TLV(X \cup Y) &= TLV(X) \cup TLV(Y) \\
TLV(X \cap Y) &= TLV(X) \cup TLV(Y) \\
TLV(c(X_1, \ldots, X_{a(c)})) &= \emptyset
\end{aligned}
$$

Definition 5.2 The *free variables* of a system of equations $\{\alpha_1 = X_1, \ldots, \alpha_n = X_n\}$ are those variables other than $\alpha_1, \ldots, \alpha_n$ that occur in $X_1, \ldots, X_n$.

Definition 5.3 A system of equations $\{\alpha_1 = X_1, \ldots, \alpha_n = X_n\}$ is in *solved form* if (1) when $\neg Y$ is a subexpression of some X_i, then Y is a free variable, and (2) $\forall i\ TLV(X_i) \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset$.

In Definition 5.3, condition (1) guarantees that there is a solution, because any substitution for the free variables can be extended (by monotonicity of the rest of the system) to a substitution satisfying the equations. Condition (2) guarantees that for any substitution for the free variables, the extension to a solution is unique. Thus, any substitution for the free variables induces a solution to the equations. The following lemma formalizes this discussion.

Lemma 5.4 Let $S = \{\alpha_1 = X_1, \ldots, \alpha_n = X_n\}$ be a system in solved form and let V be the set of all variables. For any substitution σ for the variables $V - \{\alpha_1, \ldots, \alpha_n\}$ there is a unique extension $\sigma' \in \mathcal{S}(S)$ such that $\sigma' \upharpoonright (V - \{\alpha_1, \ldots, \alpha_n\}) = \sigma$.

Proof: In appendix A.1. □

Solved form systems are essentially alternating tree automata [16]. Each variable on the left-hand side of an equation corresponds to an automaton state; the set operations of intersection and union correspond to the tree automata transitions of universal branching and existential branching respectively. Solved-form systems have one feature not found in tree automata: the possibility of unconstrained variables. Thus, solved-form systems can be regarded as tree automata with free variables.

The reduction from arbitrary systems of constraints to solved form systems has four steps; each step is explained in subsequent sections:

1. Reduce an arbitrary system of constraints to a *one-level* system (Section 6). This step puts all constraints in the form $X \subseteq 0$, where X has no unions or nested constructors and negations appear only on variables. This syntactic restriction simplifies subsequent steps and the complexity analysis.

2. Reduce a one-level system to a *cascading system* (Section 7). This step guarantees that constraints implied by constructor expressions (i.e., $c(X, Y) \subseteq 0$ implies either $X \subseteq 0$ or $Y \subseteq 0$) and transitivity (i.e., $X \subseteq Y$ and $Y \subseteq Z$ implies $X \subseteq Z$) are consistent.

3. Reduce a cascading system of constraints to a *cascading system of equations* (Section 8). In this step, constraints of the form $L \subseteq \alpha \subseteq U$ are replaced by $\alpha = L \cup (\beta \cap U)$, where β is a fresh variable. The variable β serves as a parameter allowing α to be anything "in between" the lower bound L and the upper bound U. The primary purpose of this step is change the problem from a system of constraints to a system of equations.

4. Reduce a cascading system equations to a system of equations in solved form (Section 9). This is a relatively simple step that eliminates negations and some top-level variables.

Figure 1 gives a simple but complete example of our algorithm in action. Each step is explained in subsequent sections. In this example, the two constraints $c(\alpha_2) \subseteq \neg\alpha_2$ and $c(\neg\alpha_2) \subseteq \alpha_2$ together imply that $c(x) \in \alpha_2$ iff $x \notin \alpha_2$. Thus, there are two possibilities for the meaning of α_2: $\{c^{2i}(b)\}$ or $\{c^{2i+1}(b)\}$ where i ranges over the non-negative integers.

To aid understanding of the system in Figure 1, we simplify the final result as shown in Figure 2. These simplifications are not part of the algorithm and are presented just to enhance readability. In Figure 2, note that β_1 and β_2 serve as "free variables" that allow α_1 and α_2 to be anything between the upper and lower bounds implied by the constraints. The first equation shows that α_1 is indeed a subset of α_2 and β_1 serves as a parameter identifying which subset of α_2. In the second equation, $\beta_1 \cup \beta_2$ serves as a parameter controlling whether b or $c(b)$ is an element of α_2. Finally, note that α_2 recurses on $c(c(\alpha_2))$ as desired.

initial system	constructors b, c with arities $a(b) = 0,\ a(c) = 1$ $\alpha_1 \subseteq \alpha_2$ $c(\alpha_2) \subseteq \neg\alpha_2$ $c(\neg\alpha_2) \subseteq \alpha_2$
one-level system (Section 6)	$\neg\alpha_2 \cap \alpha_1 \subseteq 0$ $\alpha_2 \cap c(\alpha_2) \subseteq 0$ $\neg\alpha_2 \cap c(\neg\alpha_2) \subseteq 0$
cascading system (Section 7)	$\neg\alpha_2 \cap \alpha_1 \subseteq 0$ $\alpha_2 \cap c(\alpha_2) \subseteq 0$ $\neg\alpha_2 \cap c(\neg\alpha_2) \subseteq 0$ $\alpha_1 \cap c(\alpha_2) \subseteq 0$
cascading equations (Section 8)	$\alpha_1 = \beta_1 \cap \neg c(\alpha_2)$ $\alpha_2 = (\beta_1 \cap \neg c(\alpha_2)) \cup c(\neg\alpha_2) \cup (\beta_2 \cap \neg c(\alpha_2))$
solved system (Section 9)	$\alpha_1 = \beta_1 \cap (b \cup c(\gamma_2))$ $\alpha_2 = (\beta_1 \cap (b \cup c(\gamma_2))) \cup c(\gamma_2) \cup (\beta_2 \cap (b \cup c(\gamma_2)))$ $\gamma_2 = (\neg\beta_1 \cup c(\alpha_2)) \cap (b \cup c(\alpha_2)) \cap (\neg\beta_2 \cup c(\alpha_2))$

Figure 1: The algorithm in action.

solved system	$\alpha_1 = \beta_1 \cap (b \cup c(\gamma_2))$ $\alpha_2 = (\beta_1 \cap (b \cup c(\gamma_2))) \cup c(\gamma_2) \cup (\beta_2 \cap (b \cup c(\gamma_2)))$ $\gamma_2 = (\neg\beta_1 \cup c(\alpha_2)) \cap (b \cup c(\alpha_2)) \cap (\neg\beta_2 \cup c(\alpha_2))$
simplify (note $\alpha_1 = \beta_1 \cap \alpha_2$)	$\alpha_1 = \beta_1 \cap \alpha_2$ $\alpha_2 = (\beta_1 \cap b) \cup (\beta_2 \cap b) \cup c(\gamma_2)$ $\gamma_2 = (\neg\beta_1 \cap \neg\beta_2 \cap b) \cup c(\alpha_2)$
eliminate variable γ_2	$\alpha_1 = \beta_1 \cap \alpha_2$ $\alpha_2 = (\beta_1 \cap b) \cup (\beta_2 \cap b) \cup c((\neg\beta_1 \cap \neg\beta_2 \cap b) \cup c(\alpha_2))$
simplify	$\alpha_1 = \beta_1 \cap \alpha_2$ $\alpha_2 = ((\beta_1 \cup \beta_2) \cap b) \cup c(\neg\beta_1 \cap \neg\beta_2 \cap b) \cup c(c(\alpha_2))$

Figure 2: Simplification of the result of Figure 1.

6 One-Level Systems of Set Constraints

A set expression is *one-level* if it has no unions or nested constructors and negations appear only on variables.

Definition 6.1 A *literal* is either a variable α or its complement $\neg\alpha$. Let $\vec{l}$ stand for a (possibly empty) conjunction of literals in which no variable appears both positively and negatively. A set expression is *one-level* if it is 0, $\vec{l} \cap 1$, or $\vec{l}_0 \cap c(\vec{l}_1 \cap 1, \ldots, \vec{l}_{a(c)} \cap 1)$.

In this section, we show that all systems of set constraints can be expressed as *one-level systems*, where the left-hand side of all constraints is a one-level expression, and the right-hand side of all constraints is 0. Using one-level systems simplifies subsequent steps and the complexity analysis.

Lemma 6.2 Let $var(S)$ be the set of all variables in a system S. There is an algorithm to compute a one-level system S' such that $\mathcal{S}(S) =_{var(S)} \mathcal{S}(S')$.

Proof: [sketch] Replace all constraints $A \subseteq B$ by $A \cap \neg B \subseteq 0$. Within expressions that are not one-level, we replace each proper subexpression E by a fresh variable α and add one-level constraints that imply $\alpha = E$. For example, we can replace $A \cap B$ by α and add constraints $\neg\alpha \cap A \cap B \subseteq 0$, $\alpha \cap \neg A \subseteq 0$, and $\alpha \cap \neg B \subseteq 0$. The complete proof is in appendix A.2. □

From this point, we assume that all systems of constraints are one-level.

7 Cascading Systems

Our algorithm for solving a system of constraints consists of a pair of transformations that convert a system to one or more equivalent systems. The transformations are applied repeatedly until either none apply or an inconsistency is detected. Our algorithm determines a system is inconsistent when it discovers that the original constraints imply either $1 \subseteq 0$ or $c \subseteq 0$ (where c is a zero-ary constructor). If no inconsistency is found, then the original system has a solution; Sections 8 through 9 prove this fact by showing how to characterize all solutions of the original system. The following definition formalizes the idea of equivalent sets of systems.

Definition 7.1 (Equivalent Systems) Let $S_1, \ldots, S_n$ and $T_1, \ldots, T_m$ be any systems of constraints. Then

$$S_1, \ldots, S_n \equiv T_1, \ldots, T_m \;\Leftrightarrow\; \bigcup_i \mathcal{S}(S_i) = \bigcup_j \mathcal{S}(T_j)$$

The first transformation enforces constraints implied by constructor expressions. It is the only transformation that may introduce multiple systems of constraints.

Rule 7.2 (Constructor Rule) $S \cup \{c(X_1, \ldots, X_n) \subseteq 0\} \equiv S_1, \ldots, S_n$ where $S_i = S \cup \{X_i \subseteq 0\}$

The Constructor Rule is correct because an expression $c(X_1, \ldots, X_n)$ is empty exactly when one of its components X_i is empty. Notice that the Constructor Rule produces one-level systems from a one-level system.

The second transformation enforces transitive constraints. Constraints $\alpha \cap X \subseteq 0$ and $\neg\alpha \cap Y \subseteq 0$ represent upper and lower bounds on α because $\alpha \cap X \subseteq 0 \Leftrightarrow \alpha \subseteq \neg X$ and $\neg\alpha \cap Y \subseteq 0 \Leftrightarrow Y \subseteq \alpha$. Intuitively, the constraints $Y \subseteq \alpha$ and $\alpha \subseteq \neg X$ have a solution only if $Y \subseteq \neg X$.

Rule 7.3 (Transitive Rule) $S \cup \{\alpha \cap X \subseteq 0, \neg\alpha \cap Y \subseteq 0\} \equiv S \cup \{\alpha \cap X \subseteq 0, \neg\alpha \cap Y \subseteq 0, X \cap Y \subseteq 0\}$

Consider the system $\{\alpha \subseteq 0, \neg\alpha \subseteq 0\}$. The Transitive Rule proves this system is inconsistent, because it adds the constraint $1 \subseteq 0$. The statement of the Transitive Rule omits an important detail; the new expression $X \cap Y$ might not be one-level. It is straightforward to transform $X \cap Y$ to a one-level expression using the following identities:

$$\begin{aligned}
x \cap \neg x &= 0 \\
0 \cap x &= 0 \\
c(\ldots, 0, \ldots) &= 0 \\
c(\ldots) \cap d(\ldots) &= 0 \text{ if } c \neq d \\
c(x_1, \ldots, x_{a(c)}) \cap c(y_1, \ldots, y_{a(c)}) &= \\
& \quad c(x_1 \cap y_1, \ldots, x_{a(c)} \cap y_{a(c)})
\end{aligned}$$

In addition to making the transitive constraint explicit, the expression $X \cap Y$ has no occurrences of α or $\neg\alpha$ at the top level (see Definition 5.1). This is a key fact, which we exploit as follows. Assign some arbitrary order $\alpha_1, \ldots, \alpha_n$ to the variables in V. We

assume from here on that all conjunctions of literals are sorted in descending order from left to right. Thus, in an expression $\alpha_j \cap X$ or $\neg\alpha_j \cap X$, j is greater than the index of any top-level variable in X. Now the largest index of any top-level variable in the constraint $X \cap Y \subseteq 0$ added by the Transitive Rule is strictly less than the largest index of any top-level variable in the original constraints.

In the final steps of the proof, we need an induction on the Herbrand level with a subinduction on the ordering of variables to show that a system is reducible to a system of equations in solved form. The next definition provides the construction used in this induction. The set of solutions $\mathcal{S}_{h,j}(S)$ is the set of substitutions that satisfy all constraints up to some Herbrand level h or $h-1$, depending on whether a constraint has a top level variable with index greater than j.

Definition 7.4

$$\begin{aligned} \mathcal{S}_{h,j}(S) &= \bigcap\{\mathcal{S}_{h,j}(X \subseteq 0) | X \subseteq 0 \in S\} \\ \mathcal{S}_{h,j}(X \subseteq 0) &= \mathcal{S}(X \cap H^{h_0} \subseteq 0) \end{aligned}$$

where $h_0 = h$ if $TLV(X) \subseteq \{\alpha_1, \ldots, \alpha_j\}$ and $h_0 = \max(0, h-1)$ otherwise.

The intuition behind the following lemma is that $\mathcal{S}_{h,j}(S)$ provides a series of increasingly accurate approximations to $\mathcal{S}(S)$. Initially, $\mathcal{S}_{0,0}(S)$ is just the set of all substitutions. As h and j increase, the set $\mathcal{S}_{h,j}(S)$ decreases monotonically, until, in the limit, it is equal to $\mathcal{S}(S)$.

Lemma 7.5

$$\begin{aligned} \mathcal{S}_{h,j}(S) &\subseteq \mathcal{S}_{h',j'}(S) \text{ if } h > h' \text{ or } h = h', j \geq j' \\ \mathcal{S}(S) &= \bigcap_{h,j} \mathcal{S}_{h,j}(S) \end{aligned}$$

Consider the system $\{\alpha_8 \subseteq 0, \neg\alpha_8 \subseteq 0\}$. Then $\mathcal{S}_{1,7}(\alpha_8 \subseteq 0, \neg\alpha_8 \subseteq 0)$ is the set of all substitutions, but $\mathcal{S}_{1,8}(\alpha_8 \subseteq 0, \neg\alpha_8 \subseteq 0)$ is the empty set. The next lemma shows that the Constructor Rule not only preserves the solution set but it also improves the approximation $\mathcal{S}_{h,j}()$.

Lemma 7.6 By applying the Constructor Rule, any one-level constraint $c(X_1, \ldots, X_n) \subseteq 0$ is equivalent a finite set of systems $\{X_1 \subseteq 0\}, \ldots, \{X_n \subseteq 0\}$ such that

$$\begin{aligned} \mathcal{S}(c(X_1, \ldots, X_n) \subseteq 0) &= \bigcup_i \mathcal{S}(X_i \subseteq 0) \\ \forall h, j\ \mathcal{S}_{h,j}(c(X_1, \ldots, X_n) \subseteq 0) &\supseteq \bigcup_i \mathcal{S}_{h,j}(X_i \subseteq 0) \end{aligned}$$

The following definition is used in Theorem 8.2 to explain how the Transitive Rule makes progress towards solving the system of constraints.

Definition 7.7 Let $\alpha_j \cap X \subseteq 0$ and $\neg\alpha_j \cap Y \subseteq 0$ be constraints in S. S is *cascading with respect to a pair of constraints* $\alpha_j \cap X \subseteq 0$ and $\neg\alpha_j \cap Y \subseteq 0$ if for all h, $\mathcal{S}_{h,j-1}(S) \subseteq \mathcal{S}_{h,j-1}(X \cap Y \subseteq 0)$. S is *cascading* if it is cascading with respect to all pairs of constraints and there are no constraints of the form $c(\ldots) \subseteq 0$ or $1 \subseteq 0$.

Consider again the system of constraints $\{\alpha_8 \subseteq 0, \neg\alpha_8 \subseteq 0\}$. This system is not cascading because $\mathcal{S}_{1,7}(\alpha_8 \subseteq 0, \neg\alpha_8 \subseteq 0) \not\subseteq \mathcal{S}_{1,7}(1 \subseteq 0)$. The proof of the next lemma gives an algorithm to transform a one-level system to a set of cascading systems.

Lemma 7.8 A one-level system S is equivalent to a finite set Γ of cascading systems.

Proof: Let $\Gamma = \{S\}$. Iterate the following four steps:

- Pick a system $S_0 \in \Gamma$ and a pair of constraints $\alpha_j \cap X \subseteq 0$ and $\neg\alpha_j \cap Y \subseteq 0$ in S_0.
- Apply the Constructor Rule (if necessary) to get a set of systems $S'_1, \ldots, S'_k \equiv \{X \cap Y \subseteq 0\}$.
- Replace S_0 in Γ by $\ldots, S_0 \cup S'_i, \ldots$
- For every $S \in \Gamma$, if S contains a constraint $1 \subseteq 0$ or $c \subseteq 0$, then delete S from Γ.

By Lemma 7.6 and Definition 7.7, $S_0 \cup S'_i$ is cascading with respect to the chosen pair of constraints. Iterating these four steps until all pairs are processed yields a finite set of cascading systems. This procedure terminates because there are only finitely many one-level expressions over a fixed set of variables and constructors. □

Let S be a system of constraints. The algorithm in the proof of Lemma 7.8 shows that S is inconsistent

if $\Gamma = \emptyset$ when the algorithm terminates. The main result of Sections 8 and 9 is that if $\Gamma \neq \emptyset$, then S has at least one solution and all solutions can be easily characterized. Thus, the proof of Lemma 7.8 is a procedure for deciding the consistency of a system of set constraints.

8 From Constraints to Equations

In the next two sections we show that cascading systems of constraints always have solutions and we show how to characterize these solutions. This section presents the most difficult step, which is to transform a cascading system of constraints to a *cascading system of equations.*

Definition 8.1 A system of equations $\{\alpha_1 = X_1, \ldots, \alpha_n = X_n\}$ is *cascading* if for each $i = 1, ..., n$ $TLV(X_i) \cap \{\alpha_i, \ldots, \alpha_n\} = \emptyset$

Recall that constraints can be viewed as upper and lower bounds on variables because $\alpha \cap X \subseteq 0 \Leftrightarrow \alpha \subseteq \neg X$ and $\neg\alpha \cap Y \subseteq 0 \Leftrightarrow Y \subseteq \alpha$. The key idea in this step is to introduce a new variable β and replace the upper and lower bounds on α by the equation $\alpha = Y \cup (\beta \cap \neg X)$. This equation guarantees that α contains at least Y and no more than $Y \cup \neg X$; the free parameter β allows the solution to be anything "in between" these upper and lower bounds.

Theorem 8.2 For any cascading system S with $var(S) = \{\alpha_1, \ldots, \alpha_n\}$ there exists a cascading system of equations S' such that $\mathcal{S}(S) =_{var(S)} \mathcal{S}(S')$.

Proof: Let $\beta_1, \ldots, \beta_n$ be new variables. Define set expressions $T_1, \ldots, T_n$ as follows.

$$T_i = (\bigcup_{\neg\alpha_i \cap X \subseteq 0 \in S} X) \cup (\beta_i \cap \bigcap_{\alpha_i \cap X \subseteq 0 \in S} \neg X)$$

Let S' be the set of equations $\{\alpha_1 = T_1, \ldots, \alpha_n = T_n\}$. Because conjunctions of literals are sorted in descending order, $TLV(T_i) \subseteq \{\beta_i, \alpha_1, \ldots, \alpha_{i-1}\}$. Thus, S' is a cascading system of equations.

To complete the proof, we must show that $\mathcal{S}(S) =_{var(S)} \mathcal{S}(S')$. We first show $\mathcal{S}(S) \subseteq_{var(S)} \mathcal{S}(S')$. Let $\sigma \in \mathcal{S}(S)$ and let $\gamma = \sigma[\ldots, \beta_i \leftarrow \sigma(\alpha_i), \ldots]$. Clearly $\gamma =_{var(S)} \sigma$. It suffices to prove that each equation $\alpha_i = T_i$ holds under the substitution γ.

$$\begin{aligned} T_i &= \bigcup_{\neg\alpha_i \cap X \subseteq 0 \in S} X \cup (\beta_i \cap \bigcap_{\alpha_i \cap X \subseteq 0 \in S} \neg X) \\ &= \bigcup_{\neg\alpha_i \cap X \subseteq 0 \in S} X \cup (\alpha_i \cap \bigcap_{\alpha_i \cap X \subseteq 0 \in S} \neg X) \\ &= \bigcup_{\neg\alpha_i \cap X \subseteq 0 \in S} X \cup \alpha_i \\ &= \alpha_i \end{aligned}$$

In this proof, the first equation is the definition of T_i. The second equation follows because $\mu(\alpha_i, \gamma) = \mu(\beta_i, \gamma)$. The third equation follows because $\alpha_i \subseteq \neg X$ for each $\alpha_i \cap X \subseteq 0 \in S$. The last step follows because $X \subseteq \alpha_i$ for each $\neg\alpha_i \cap X \subseteq 0 \in S$.

For the other direction, we must show that $\mathcal{S}(S') \subseteq_{var(S)} \mathcal{S}(S)$. Let σ be a substitution satisfying the equations. We show by induction on (h, i) that $\sigma \in \mathcal{S}_{h,i}(X \subseteq 0)$ for every constraint $X \subseteq 0$ in S. The base case $(h, i) = (0, 0)$ is trivial, since $X \cap H^0 = 0$. Assume the result holds for $(h, i-1)$. We break the inductive step into two cases: one case for constraints of the form $\alpha_j \cap Y \subseteq 0$, and one case for constraints of the form $\neg\alpha_j \cap Y \subseteq 0$. For a constraint $\alpha_j \cap Y \subseteq 0$, note $\mathcal{S}_{h,i}(\alpha_j \cap Y \subseteq 0) = \mathcal{S}_{h,i-1}(\alpha_j \cap Y \subseteq 0)$ holds if $i \neq j$. If $i = j$, then

$$\begin{aligned} & \alpha_j \cap Y \\ = \; & T_j \cap Y \\ = \; & (\bigcup_{\neg\alpha_j \cap X \subseteq 0 \in S} X \cup (\beta_j \cap \bigcap_{\alpha_j \cap X \subseteq 0 \in S} \neg X)) \cap Y \\ \subseteq \; & (\bigcup_{\neg\alpha_j \cap X \subseteq 0 \in S} X \cup (\beta_j \cap \neg Y)) \cap Y \\ = \; & \bigcup_{\neg\alpha_j \cap X \subseteq 0 \in S} (X \cap Y) \\ = \; & 0 \end{aligned}$$

In this proof, the first equation follows because $\mu(\alpha_j, \sigma) = \mu(T_j, \sigma)$. The second equation holds by definition of T_j. The third step follows because $\alpha_j \cap Y \subseteq 0 \in S$. The last equation follows by induction and the cascading property; this deserves more explanation. By the inductive hypothesis $\sigma \in \mathcal{S}_{h,j-1}(S)$, and for any constraint $\neg\alpha_j \cap X \subseteq 0$, $\mathcal{S}_{h,j-1}(S) \subseteq \mathcal{S}_{h,j-1}(Y \cap X \subseteq 0)$ by the cascading property. Therefore, $\mu(Y \cap X \cap H^h, \sigma) = 0$ since $TLV(Y \cap X) \subseteq \{\alpha_1, \ldots, \alpha_{j-1}\}$. Checking constraints

of the form $\neg\alpha_j \cap Y \subseteq 0$ does not depend on the inductive hypothesis:

$$\begin{aligned}
& \neg\alpha_j \cap Y \\
= \;& \neg T_j \cap Y \\
= \;& \neg\Big(\bigcup_{\neg\alpha_j \cap X \subseteq 0 \in S} X \cup (\beta_j \cap \bigcap_{\alpha_j \cap X \subseteq 0 \in S} \neg X)\Big) \cap Y \\
= \;& \Big(\bigcap_{\neg\alpha_j \cap X \subseteq 0 \in S} \neg X \cap (\neg\beta_j \cup \bigcup_{\alpha_j \cap X \subseteq 0 \in S} X)\Big) \cap Y \\
\subseteq \;& (\neg Y \cap (\neg\beta_j \cup \bigcup_{\alpha_j \cap X \subseteq 0 \in S} X)) \cap Y \\
= \;& 0
\end{aligned}$$

The first equation follows because $\mu(\alpha_j, \sigma) = \mu(T_j, \sigma)$. The second equation follows by definition of T_j. The third equation is a simplification; the fourth step follows because $\neg\alpha_j \cap Y \subseteq 0 \in S$. □

9 The Final Step

For the final step, we show how to reduce a cascading system of equations to a system of equations in solved form. This involves two relatively simple transformations. First, we must ensure that all the top-level variables of equations are free variables. Second, we must eliminate negations, except where they occur on free variables.

Let $\{\alpha_1 = E_1, \ldots, \alpha_n = E_n\}$ be a cascading system of equations. Because the system is cascading, E_i has no top-level occurrences of the variables $\alpha_i, \ldots, \alpha_n$. To eliminate the remaining top-level variables, simply replace, in order, α_j by E_j at the top level on every right-hand side for $1 \leq j \leq n$. In the resulting system, all top-level variables are free variables.

Removing negations is accomplished by an algorithm that drives negations inside until they occur only on variables. Given an expression E, the *negation normal form* $NNF(E)$ of E is the expression produced by top-down application of the following transformations to E:

$$\begin{aligned}
\neg 0 &= 1 \\
\neg 1 &= 0 \\
\neg(\neg X) &= X \\
\neg(X \cup Y) &= \neg X \cap \neg Y \\
\neg(X \cap Y) &= \neg X \cup \neg Y \\
\neg c(X_1, \ldots, X_{a(c)}) &= \bigcup_{d \in C - \{c\}} d(1, \ldots, 1) \cup \\
&\qquad \bigcup_{1 \leq i \leq a(c)} c(\ldots, 1, \neg X_i, 1, \ldots)
\end{aligned}$$

Lemma 9.1 Let $\{\alpha_1 = X_1, \ldots, \alpha_n = X_n\}$ be a cascading system of equations with variables V, let $\gamma_1, \ldots, \gamma_n$ be distinct variables not in V, and let S1 and S2 be systems of equations as defined below, where i and j range from 1 to n. Then $\mathcal{S}(S1) =_V \mathcal{S}(S2)$.

$$\begin{aligned}
\text{S1:}\quad \alpha_i &= X_i \\
\text{S2:}\quad \alpha_i &= NNF(X_i)[\ldots\gamma_j/\neg\alpha_j\ldots] \\
\gamma_i &= NNF(\neg X_i)[\ldots\gamma_j/\neg\alpha_j\ldots]
\end{aligned}$$

To finish the reduction, we observe that if all top-level variables of the cascading system of equations in Lemma 9.1 are free variables, then the system $S2$ is in solved form. An immediate corollary is that if the algorithm of Lemma 7.8 does not report that a system of set constraints is inconsistent, then the system has a solution.

10 Complexity Analysis

We show that the decision problem of determining whether a system of set constraints is consistent (has a solution) is hard for exponential time, and that it is in nondeterministic exponential time. The exact complexity remains an open problem.

Theorem 10.1 The consistency problem for systems of set constraints is EXPTIME-hard.

Proof: [sketch] In [15] it is shown that determining whether two *finite tree automata* accept the same language is complete for EXPTIME. Finite tree automata are equivalent to solved-form systems of equations with no free variables. More precisely, for every finite tree automaton A there is a solved-form system of equations $S = \{\alpha_i = E_i\}$, computable in time polynomial in the size of A, such that the language $\mathcal{L}(A)$ accepted by A is $\mu(\alpha_1, \sigma)$, where σ is the unique solution of S.

To reduce this problem to solving systems of equations, for any two finite tree automata A and B, let $\{\alpha_i = E_i\}$ and $\{\beta_j = E'_j\}$ be corresponding solved-form systems without free variables where the α_i and

β_j are disjoint. Then the system $\{\alpha_i = E_i, \beta_j = E'_j, \alpha_1 = \beta_1\}$ has a solution if and only if $\mathcal{L}(A) = \mathcal{L}(B)$. □

Theorem 10.2 The consistency problem for systems of set constraints is in NEXPTIME.

Proof: [sketch] Consider a set of constraints with m variables, p constructors, and maximum constructor arity k. The proof is divided into three steps. We first prove the result for one-level systems with maximum constructor arity $k = 2$. We then extend the theorem to arbitrary systems where $k = 2$. Finally, we sketch how the proof is extended to systems where constructors have arbitrary arity. Let n be the total size (the number of symbols) of the system.

Each step of the algorithm in the proof of Lemma 7.8 adds one one-level expression to the system of constraints. Therefore, a simple way to bound the complexity of this algorithm is to count the number of possible one-level expressions and multiply by the time it takes to perform one step. There are 3^m possible conjunctions of literals, as each literal may be positive, negative, or absent. When $k = 2$, the largest one-level expression has the form $\vec{l}_0 \cap c(\vec{l}_1 \cap 1, \vec{l}_2 \cap 1)$. Thus, there are at most about $p(3^{3m})$ one-level expressions, which is $2^{\mathcal{O}(n)}$.

One iteration of the procedure of Lemma 7.8 takes at most $2^{\mathcal{O}(n)}$, because there are at most $(2^n)^2 = 2^{2n}$ pairs of constraints to consider and each step (considering one pair of constraints) takes time at most polynomial in n. When applying the Constructor Rule (7.2), one of the possible sets of constraints is chosen nondeterministically. The procedure of Lemma 7.8 halts in at most $2^{\mathcal{O}(n)}$ iterations, since either the system is found to be inconsistent or it is eventually cascading with respect to all pairs of constraints, and there are at most $2^{\mathcal{O}(n)}$ constraints. Thus, the overall time is $2^{\mathcal{O}(n)}$. To finish the proof, note that a cascading system always has a solution.

Now consider any system of constraints with maximum constructor arity $k = 2$. From the proof of Lemma 6.2 (see Section A.2), any system can be reduced to a one-level system with $\mathcal{O}(n)$ variables in time polynomial in n . From above, a one-level system with $\mathcal{O}(n)$ variables and $k = 2$ can be solved in nondeterministic time $2^{\mathcal{O}(n)}$.

Finally, let S be an arbitrary system of constraints with constructors C and variables V. The following algorithm converts S into a system S' such that S' has a solution if and only if S does and the maximum arity of constructors in S' is 2. The idea is to replace n-ary constructors in S by a sequence of nested binary constructors. For each constructor c in S where $a(c) > 2$ let c' be a new constructor. We will also need one additional constructor d and a fresh variable γ. We add the following constraints to S:

$$\begin{aligned} \gamma &= \bigcup_{c \in C} c(\gamma, \ldots, \gamma) \\ \alpha &\subseteq \gamma \text{ for every } \alpha \in V \end{aligned}$$

These constraints do not change the solutions of S because γ is the entire Herbrand Universe. The purpose of these constraints is explained below. The function h converts an expression in S to an expression with constructors of arity at most 2.

$$\begin{aligned} h(0) &= 0 \\ h(1) &= \gamma \\ h(\alpha) &= \alpha \\ h(\neg X) &= \gamma \cap \neg h(X) \\ h(X \cup Y) &= h(X) \cup h(Y) \\ h(X \cap Y) &= h(X) \cap h(Y) \\ h(c(X_1, \ldots, X_{a(c)})) &= \\ & \quad c(h(X_1), \ldots, h(X_{a(c)})) \text{ if } a(c) \leq 2 \\ h(c(X_1, \ldots, X_{a(c)-1}, X_{a(c)})) &= \\ & \quad c'(h(X_1), d(\ldots, d(X_{a(c)-1}, X_{a(c)}) \ldots)) \end{aligned}$$

Let $S' = \{h(X) \subseteq h(Y) | X \subseteq Y \in S\}$. Every solution $\sigma \in \mathcal{S}(S)$ induces a solution $\sigma' \in \mathcal{S}(S')$ where $\sigma'(\alpha) = \{h(t) | t \in \sigma(\alpha)\}$. In the other direction, if $\sigma' \in \mathcal{S}(S')$, then $\sigma \in \mathcal{S}(S)$ where $\sigma(\alpha) = \{h^{-1}(t) | t \in \sigma'(\alpha)\}$. The function h^{-1} is well-defined because the constraints $\alpha \subseteq \gamma$ guarantee that the solutions of S' assign only subsets of γ to the variables, and the meaning of γ in S' is the image under h of the Herbrand Universe of S. Thus, S' has a solution if and only if S does. The algorithm h takes linear time in the size of S, so from the above discussion the complexity of deciding whether an arbitrary system of constraints has a solution is in NEXPTIME. □

11 Future Work

There are two outstanding problems we intend to pursue. The first is to prove a tighter bound on the complexity of solving systems of set constraints.

It seems unlikely that a more efficient algorithm exists; we conjecture that the problem is complete for NEXPTIME.

The second problem is to extend the algorithm to handle a wider class of set constraints. In particular, we would like to extend these techniques to projection functions.

Acknowledgements

We would like to thank Moshe Vardi for extensive discussions, and Joe Halpern, Moshe Vardi, Jennifer Widom, and John Williams for their comments on earlier drafts of this paper.

A Appendix

This appendix includes proofs omitted from the body of the paper.

A.1 Solved Form Systems

The following proof of Lemma 5.4 is based on techniques in [12].

Proof: The function μ is monotonic in all set expression operations except for set complement. Let σ be any substitution for the variables $V - \{\alpha_1, \ldots, \alpha_n\}$. By assumption, the only negations appear on free variables. Thus, by the monotonicity of the rest of the system, there is a least extension σ' of σ that satisfies the equations. Let γ be any extension of σ that solves the equations. We show that $\mu(\alpha_i \cap H^k, \sigma') = \mu(\alpha_i \cap H^k, \gamma)$ for all k; it follows that $\sigma' = \gamma$. For the base case, $\mu(\alpha_i \cap H^0, \sigma') = 0 = \mu(\alpha_i \cap H^0, \gamma)$.

Assume that $\mu(\alpha_i \cap H^{k-1}, \sigma') = \mu(\alpha_i \cap H^{k-1}, \gamma)$. Now α_i is the left-hand side of an equation $\alpha_i = E_i(\alpha_1, \ldots, \alpha_n)$ where no α_j occurs at the top level; i.e., on the right-hand side every α_i appears inside a constructor. Therefore, $E_i(\alpha_1, \ldots, \alpha_n) \cap H^k$ equals $E_i(\alpha_1 \cap H^{k-1}, \ldots, \alpha_n \cap H^{k-1}) \cap H^k$. Using this fact we can prove by induction:

$$\begin{aligned} & \mu(\alpha_i \cap H^k, \sigma') \\ = \; & \mu(E_i(\alpha_1, \ldots, \alpha_n) \cap H^k, \sigma') \\ = \; & \mu(E_i(\alpha_1 \cap H^{k-1}, \ldots, \alpha_n \cap H^{k-1}) \cap H^k, \sigma') \\ = \; & \mu(E_i(\alpha_1 \cap H^{k-1}, \ldots, \alpha_n \cap H^{k-1}) \cap H^k, \gamma) \\ = \; & \mu(E_i(\alpha_1, \ldots, \alpha_n) \cap H^k, \gamma) \\ = \; & \mu(\alpha_i \cap H^k, \gamma) \end{aligned}$$

□

A.2 Reduction to One-Level Systems

In this section we prove that any system of constraints is reducible to a one-level system (Lemma 6.2).

Proof: Replace all constraints $A \subseteq B$ by $A \cap \neg B \subseteq 0$. Within expressions that are not one-level, we replace each subexpression E by a fresh variable α and add one-level constraints that imply $\alpha = E$. More formally, for each left-hand side of a constraint that is not one-level, apply the following transformations bottom-up. Upon termination, replace conjunctions of literals $\vec{l}$ by $\vec{l} \cap 1$ as necessary to guarantee that expressions are one-level. In each transformation, γ is a fresh variable.

- Replace 0 by γ and add constraint $\gamma \subseteq 0$.
- Replace 1 by γ and add constraint $\neg\gamma \subseteq 0$.
- Replace $\alpha \cap \beta$ by γ and add constraints $\neg\gamma \cap \alpha \cap \beta \subseteq 0$, $\gamma \cap \neg\alpha \subseteq 0$, $\gamma \cap \neg\beta \subseteq 0$.
- Replace $\alpha \cup \beta$ by γ and add constraints $\neg\gamma \cap \alpha \subseteq 0$, $\neg\gamma \cap \beta \subseteq 0$, $\gamma \cap \neg\alpha \cap \neg\beta \subseteq 0$.
- Replace $\neg\alpha$ by γ and add constraints $\gamma \cap \alpha \subseteq 0$, $\neg\gamma \cap \neg\alpha \subseteq 0$.
- Replace $c(\alpha_1, \ldots, \alpha_n)$ by γ and add constraints $\neg\gamma \cap c(\alpha_1, \ldots, \alpha_n) \subseteq 0$ and $\forall 1 \leq i \leq n \quad \gamma \cap c(1, \ldots, 1, \neg\alpha_i, 1, \ldots, 1) \subseteq 0$, $\forall d \neq c \;\; \gamma \cap d(1, \ldots, 1) \subseteq 0$.

Note that all the added constraints are one-level (assuming $\vec{l} \cap 1$ is substituted for $\vec{l}$). It is easy to show that these transformations preserve the solutions of the original system; we prove this only for the case $\alpha \cap \beta$. First, $\neg\gamma \cap \alpha \cap \beta \subseteq 0$ implies that $\alpha \cap \beta \subseteq \gamma$. For the other direction, $\gamma \cap \neg\alpha \subseteq 0 \Rightarrow \gamma \subseteq \alpha$ and $\gamma \cap \neg\beta \subseteq 0 \Rightarrow \gamma \subseteq \beta$ together imply that $\gamma \subseteq \alpha \cap \beta$. So, $\gamma = \alpha \cap \beta$.

Let n be the size of the original system of constraints. The time complexity of this algorithm is polynomial in n. Also, the algorithm introduces at most $\mathcal{O}(n)$ new variables. □

References

[1] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[2] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, January 1991.

[3] J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.

[4] Gilberto Filé. Tree automata and logic programs. In *Second Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, January 1985. Lecture Notes in Computer Science 182.

[5] T. Früwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Symposium on Logic in Computer Science*, pages 300–309, July 1991.

[6] F. Gecseg and M. Steinby. *Tree Automata*. Academei Kaido, Budapest, 1984.

[7] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[8] 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.

[9] N. Heintze and J. Jaffar. Set-based program analysis. Draft manuscript, 1991.

[10] 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.

[11] P. Mishra. Towards a theory of types in PROLOG. In *Proceedings of the First IEEE Symposium in Logic Programming*, pages 289–298, 1984.

[12] P. Mishra and U. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21, 1985.

[13] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, (141):1–35, 1969.

[14] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[15] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[16] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[17] J. Young and P. O'Keefe. Experience with a type evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581. North-Holland, 1988.