



First-Order Theory of Subtyping Constraints

Zhendong Su, Alex Aiken, Joachim Niehren, Tim Priesnitz, Ralf Treinen

► **To cite this version:**

Zhendong Su, Alex Aiken, Joachim Niehren, Tim Priesnitz, Ralf Treinen. First-Order Theory of Subtyping Constraints. The 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2002, Portland, United States. pp.203-216. inria-00536828

HAL Id: inria-00536828

<https://hal.inria.fr/inria-00536828>

Submitted on 16 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The First-Order Theory of Subtyping Constraints

Zhendong Su* Alexander Aiken*

Computer Science Division
University of California
Berkeley, CA 94720

<http://www.cs.berkeley.edu/~{zhendong,aiken}>

Joachim Niehren Tim Priesnitz

Programming Systems Lab
Universität des Saarlandes
Saarbrücken, Germany

<http://www.ps.uni-sb.de/~{niehren,tim}>

Ralf Treinen

LRI, Université Paris-Sud
F91405 Orsay cedex, France

<http://www.lri.fr/~treinen>

Abstract

We investigate the first-order theory of subtyping constraints. We show that the first-order theory of non-structural subtyping is undecidable, and we show that in the case where all constructors are either unary or nullary, the first-order theory is decidable for both structural and non-structural subtyping. The decidability results are shown by reduction to a decision problem on tree automata. This work is a step towards resolving long-standing open problems of the decidability of entailment for non-structural subtyping.

1 Introduction

In this paper we present the first decidability and undecidability results for the first-order theory of subtyping. Before describing our results, we begin with a capsule history of subtyping, which motivates the first-order theory of subtyping as an interesting topic to study.

Since the original results of Mitchell [24], type checking and type inference for subtyping systems have received steadily increasing attention. The primary motivations for studying these systems today are program analysis algorithms based on subtyping (see, for example, [2, 5, 14, 16, 22, 31, 37]) and, more speculatively, richer designs for typed languages ([29]).

Subtyping algorithms invariably involve systems of subtyping constraints $\tau_1 \leq \tau_2$, where the τ_i are types that may

*This research was supported in part by the National Science Foundation grant No. CCR-0085949 and NASA Contract No. NAG2-1210.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL '02, Jan. 16-18, 2002 Portland, OR USA
© 2002 ACM ISBN 1-58113-450-9/02/01...\$5.00

contain type variables. There are two interesting questions we can ask about a system of subtyping constraints C :

1. Does C have solutions (and what are they)?
2. Does C imply (or *entail*) another system of constraints C' ? That is, is every solution of C also a solution of C' ?

For (1), the basic algorithms for solving many natural forms of subtyping constraints are by now quite well understood (e.g., see [36]). For (2), there has been much less progress on subtype entailment, although entailment is as important as constraint resolution in applications of subtyping. For example, a type-based program analysis extracts some system of constraints C from a program text; these constraints are the model of whatever program property is being analyzed. A client of the analysis (e.g., a program optimization system) interacts with the analysis by asking queries: Does a particular constraint $\tau_1 \leq \tau_2$ hold in C ? Or in other words, does C entail $\tau_1 \leq \tau_2$? As another example, in designing a language with expressive subtyping relationships, checking type interfaces also reduces to a subtype entailment problem. While no mainstream language has such expressive power today, language researchers have encountered just this problem in designing languages that blend ML-style polymorphism with object-oriented style subtyping, which leads to *polymorphic constrained types* (see, again, discussion in [29]).

There are two natural choices of subtype relation in the literature. *Structural subtyping* requires that types have exactly the same shape—read as trees, $\tau_1 \leq \tau_2$ cannot hold unless the corresponding branches of τ_1 and τ_2 are equal in length. For example if $a \leq b$ in the subtype ordering, then $C(a, a) \leq C(b, b)$ for some covariant constructor C , but $a \not\leq C(b, b)$. *Non-structural* subtyping has both a least type \perp and a greatest type \top , so that $\perp \leq \tau \leq \top$ for any τ . More details on structural and non-structural subtyping can be found in [4, 19, 24].

Despite extensive effort over many years, the exact complexity and even the decidability of entailment is open for non-structural subtyping constraints [3, 12, 13, 17, 18, 22, 27, 28, 33, 34, 36, 42]. As we show in Section 2, the natural versions of entailment and subtyping constrained types can

be encoded easily in the first-order theory of subtyping, so to gain insight into and take a step towards resolving these difficult problems, we study the full first-order theory in this paper.

The major contributions of this paper are summarized as follows:

- We show that the first-order theory of non-structural subtyping constraints is undecidable via a reduction from the Post’s Correspondence Problem (PCP). The result is shown for both simple and recursive types (Theorems 7 and 9). The result holds also for infinite trees. In addition, this result yields a technical separation of structural subtyping and non-structural subtyping (Theorem 10).
- We show that the first-order theory of subtyping constraints with unary function symbols is decidable by an automata-theoretic construction. This result holds for all combinations of the structural versus non-structural, and simple versus recursive cases (Theorem 17).
- The automata-theoretic construction bridges tree automata theory and subtyping problems, suggesting an alternative way of tackling the problems (see Section 5.3 for a discussion).

We leave open the decidability of the full first-order theory of structural subtyping.

We first present background information on subtyping (Section 2), and show that the first-order theory of non-structural subtyping entailment is undecidable (Section 3). Next we give an automata-theoretic construction for subtyping constraints and show that the first-order theory of subtyping constraints with unary function symbols is decidable (Section 5). We then discuss related work (Section 6) and conclude (Section 7). An example encoding of an entailment problem is given in Appendix A.

2 Subtyping Constraints and Their First-Order Theories

We present an overview of subtyping systems and introduce the problems we consider in this paper.

2.1 Preliminaries on Subtyping

Subtyping systems are generalizations of the usual equality-based type systems such as the Hindley/Milner type system of ML [23]. We consider the following type language

$$\tau ::= \perp \mid \top \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2$$

where \perp and \top are the smallest and largest type respectively, α is chosen from a countable set of type variables \mathcal{V} , \rightarrow is the function type constructor, and \times is the product type constructor.

Types in this language form a lattice with the following ordering

- $\perp \leq \tau \leq \top$, for any τ ;
- $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$ iff $\tau'_1 \leq \tau_1$ and $\tau_2 \leq \tau'_2$, for any types τ_1, τ_2, τ'_1 , and τ'_2 ;
- $\tau_1 \times \tau_2 \leq \tau'_1 \times \tau'_2$ iff $\tau_1 \leq \tau'_1$ and $\tau_2 \leq \tau'_2$, for any types τ_1, τ_2, τ'_1 , and τ'_2 .

This is the *non-structural ordering* on types, since related types need not have the same shape, e.g., $\perp \leq \perp \rightarrow \top$. The corresponding notion of *structural ordering* requires two types to be related only if they have the same shape. In structural ordering, there is no smallest or largest type.

Another dimension is whether a type language allows *recursive types*, i.e., infinite types which are solutions to recursive type equations such as $\alpha = \alpha \rightarrow \perp$. Recursive types are interpreted over *regular trees*, which are possibly infinite trees with finitely many subterms. We also consider general infinite trees.

We write $T(\mathcal{F})$ to denote the set of finite *ground types* (types without variables), where \mathcal{F} is the alphabet

$$\{\perp, \top, \cdot \rightarrow \cdot, \cdot \times \cdot\}$$

The set $T(\mathcal{F}, \mathcal{V})$ denotes the set of all types built with variables drawn from \mathcal{V} .

A *subtype constraint* is an inequality of the form $\tau_1 \leq \tau_2$. A *subtype constraint system* is a finite set of subtyping constraints. When clear from context, we drop “subtype” and just say a constraint or a constraint system. For a constraint system C , the type variables in C are called the *free variables* of C , denoted $\text{fv}(C)$.

A *valuation* ρ is a function mapping type variables \mathcal{V} to ground types $T(\mathcal{F})$. A valuation ρ is sometimes referred to as a ground substitution. As is standard, we extend valuations homomorphically to substitutions from $T(\mathcal{F}, \mathcal{V})$ to $T(\mathcal{F})$.

A valuation ρ *satisfies* a constraint $\tau_1 \leq \tau_2$, written $\rho \models \tau_1 \leq \tau_2$ if $\rho(\tau_1) \leq \rho(\tau_2)$ holds in the lattice $T(\mathcal{F})$. A valuation ρ *satisfies* a constraint system C , written $\rho \models C$, if ρ satisfies all the constraints in C . A constraint system C is *satisfiable* if there is a valuation ρ such that $\rho \models C$. The set of valuations satisfying a constraint system C is the *solution set* of C , denoted by $S(C)$. We denote by $S(C)|_E$ the set of solutions of C restricted to a set of variables E . The *satisfiability problem* for a constraint language is to decide whether a given system of constraints is satisfiable. It is well-known that the satisfiability of a constraint system can be decided in polynomial time by a test for *consistency* of the given constraint set according to a set of syntactic rules [20, 30, 33].

Corresponding to polymorphic type schemes in Hindley/Milner style type systems, polymorphic subtype systems have so-called *constrained types*, in which a type is restricted by a system of constraints [1, 3, 42]. An ML style polymorphic type can be viewed as a constrained type with no constraints. For example,

$$\alpha \rightarrow \beta \setminus \{\alpha \leq \mathbf{int} \rightarrow \mathbf{int}, \mathbf{int} \rightarrow \alpha \leq \beta\}$$

is a constrained type. Let $\tau \setminus C$ be a constrained type, and let ρ be a satisfying valuation for C . The ground type $\rho(\tau)$ is called a *instance* of $\tau \setminus C$.

There are a few important problems associated with constrained types in polymorphic subtype systems.

- In practice, constrained types can be large and complicated. Thus it is important to simplify the types [12, 22, 33] to make the types and the associated constraints smaller. Type and constraint simplification is related to the following decision problem of *constraint entailment*: A constraint system C *entails* a constraint $\tau_1 \leq \tau_2$, written $C \models \tau_1 \leq \tau_2$, if for every satisfying valuation ρ of C , we have $\rho \models \tau_1 \leq \tau_2$.

- The notion of *existential entailment*, written $C_1 \vDash \exists E.C_2$, is a more powerful notion of entailment.¹ The entailment holds if for every valuation $\rho \vDash C_1$, there exists a valuation $\rho' \vDash C_2$ such that ρ and ρ' agree on variables $\text{fv}(C_2) \setminus E$. We assume w.l.o.g. that $\text{fv}(C_1) \cap E = \emptyset$. This notion is interesting because usually for a constrained type, we are only interested in variables appearing in the type, and there are often many “internal” variables in the constraints we may wish to eliminate. This notion of entailment allows more powerful simplification and is likely to be more expensive.
- In polymorphic subtype systems, we may need to determine whether one constrained type is a subtype of another constrained type [42]. Let $\tau_1 \setminus C_1$ and $\tau_2 \setminus C_2$ be two constrained types. We wish to check whether $\tau_1 \setminus C_1 \leq \tau_2 \setminus C_2$ which holds if for every instance τ of $\tau_2 \setminus C_2$, there exists an instance τ' of $\tau_1 \setminus C_1$ such that $\tau' \leq \tau$. We assume w.l.o.g. that C_1 and C_2 do not have any variables in common. In addition, we can restrict τ_1 and τ_2 to variables because

$$\tau_1 \setminus C_1 \leq \tau_2 \setminus C_2$$

iff

$$\alpha \setminus (C_1 \cup \{\alpha = \tau_1\}) \leq \beta \setminus (C_2 \cup \{\beta = \tau_2\})$$

where α and β are fresh variables not in C_1 or C_2 .

Although extensive research has directed at these problems [3, 12, 13, 17, 18, 22, 27, 28, 33, 34, 36, 42], their decidability has been open for many years. In this paper, we present results on the first-order theory of subtyping constraints, which we believe is a step in resolving these open problems.

2.2 First-Order Theory of Subtyping Constraints

We first define the *first-order theory of subtyping constraints*. First-order formulae w.r.t. to a subtype language are:

$$f ::= \text{true} \mid t_1 \leq t_2 \mid \neg f \mid f_1 \wedge f_2 \mid \exists x.f$$

where t_1 and t_2 are type expressions and x is a first-order variable ranging over types. Notice that we do not need equality because \leq is anti-symmetric.

As usual, for convenience, we also allow disjunction \vee , implication \rightarrow , and universal quantification \forall . We write $t_1 \not\leq t_2$ for $\neg(t_1 \leq t_2)$. A formula is *quantifier free* if it has no quantifiers. A formula is in *prenex normal form* if it is of the form $Q_1 \dots Q_n.f$ where Q_i 's are quantifiers and f is a quantifier free formula. We adopt the usual notion of a *free variable* and a *closed* and *open* formula.

We next show how the open entailment problems discussed in Section 2.1 fit in the first-order theory of subtyping.

2.2.1 Entailment is in the \forall -Fragment

The universal fragment consists of all the closed formulae $\forall.f$, where \forall consists of a set of universal quantifiers, and f is a quantifier free formula.

The entailment problem $C \vDash x \leq y$ is in the universal fragment. Notice that C is a conjunction of basic constraints

¹Existential entailment is also called *restricted entailment*, written $C_1 \vDash_{E'} C_2$, where $E' = \text{fv}(C_2) \setminus E$.

and the entailment $C \vDash x \leq y$ holds iff the universal formula $\forall x_1, \dots, x_n.(C \rightarrow (x \leq y))$ is valid, where the x_i 's are the variables free in $C \cup \{x \leq y\}$.

2.2.2 Existential Entailment is in the $\forall\exists$ -Fragment

The $\forall\exists$ -fragment consists of all the closed formulae $\forall\exists.f$, where f is a quantifier free formula.

Existential entailment $C_1 \vDash \exists E.C_2$ is expressed by the following formula:

$$\forall \alpha_1, \dots, \alpha_n.(C_1 \rightarrow \exists E.C_2)$$

where the α_i 's are the variables in $\text{fv}(C_1) \cup (\text{fv}(C_2) \setminus E)$. Because we assume $\text{fv}(C_1) \cap E = \emptyset$, there is an equivalent formula in the $\forall\exists$ -fragment

$$\forall \alpha_1, \dots, \alpha_n.\exists E.(C_1 \rightarrow C_2)$$

2.2.3 Subtype Constrained Types is in the $\forall\exists$ -Fragment

Let $\alpha \setminus C_1$ and $\beta \setminus C_2$ be constrained types. We express $\alpha \setminus C_1 \leq \beta \setminus C_2$ as the formula

$$\forall \beta_1, \dots, \beta_n.(C_2 \rightarrow \exists \alpha_1, \dots, \alpha_m.(C_1 \wedge \alpha \leq \beta))$$

where the α_i 's and β_j 's are the variables free in C_1 and C_2 respectively. Because C_1 and C_2 have disjoint sets of variables (see definition of constrained types above), this is equivalent to

$$\forall \beta_1, \dots, \beta_n.\exists \alpha_1, \dots, \alpha_m.(C_2 \rightarrow (C_1 \wedge \alpha \leq \beta))$$

In fact, we can show that subtype constrained types can be polynomially reduced to existential entailment.

Proposition 1 Subtype constrained types is polynomially reducible to existential entailment.

Proof. We have the following equivalences

$$\alpha \setminus C_1 \leq \beta \setminus C_2$$

$$\Leftrightarrow \{ \text{by defn. of } \alpha \setminus C_1 \leq \beta \setminus C_2 \}$$

$$S(C_2)|_{\{\beta\}} \subseteq S(\alpha \leq \beta \wedge C_1)|_{\{\beta\}}$$

$$\Leftrightarrow \{ \text{by defn. of existential entailment with } E = \text{fv}(C_1) \}$$

$$C_2 \vDash \exists E.(\alpha \leq \beta \wedge C_1)$$

□

3 Undecidability of the First-Order Theory

In this section, we show that the first-order theory of non-structural subtyping is undecidable for any type language with a binary type constructor and the bottom element \perp (or dually, the top element \top). The formula we exhibit is in the $\exists\forall\exists\forall\exists\forall\exists$ -fragment.

The proof is via a reduction from the Post's Correspondence Problem (PCP) [32] to a first-order formula of non-structural subtyping. Since PCP is undecidable [32], the first-order theory of non-structural subtyping is undecidable as well. The proof follows the framework of Treinen [40] and is inspired by the proof of undecidability of the first-order theory of ordering constraints over feature trees [25].

Recall that an instance of PCP is a finite set of pairs of words $\langle l_i, r_i \rangle$ for $1 \leq i \leq n$. The words are drawn from the alphabet $\{1, 2\}$. The problem is to decide whether there is a non-empty finite sequence of indices $s_1 \dots s_m$ (where $1 \leq s_i \leq n$ for $1 \leq i \leq m$) and the sequence constitutes a pair of matched words:

$$l_{s_1} \dots l_{s_m} = r_{s_1} \dots r_{s_m}$$

where words are concatenated.

For non-structural subtyping, we consider both finite types and recursive types. We first describe the subtype logic that we use. We consider any subtype language with at least a bottom element \perp and a binary type constructor. We show that for any such language, the first-order theory of non-structural subtype entailment is undecidable.

For the rest of the paper, we consider the simple expression language:

$$\tau ::= \perp \mid f(\tau, \tau)$$

where f is covariant in both of its arguments. It is straightforward to modify our construction to allow type constructors with contravariant field(s) and with arity greater than two.

3.1 Representing Words as Trees

PCP is a word problem but types are trees. As a first step, we describe how to encode words in $\{1, 2\}$ using types.

3.1.1 Words as f -Spines

We first describe how to represent words over $\{1, 2\}$ as trees over a binary constructor f and the constant \perp . We use f -spines to represent words. Intuitively, an f -spine is simply a tree with a spine of f 's and all other positions labelled \perp .

Definition 2 (f -Spine) A finite tree t (in f and \perp) is an f -spine if there is *exactly one maximal path* with labels f . On this maximal path, a left child represents 1 and a right child represents 2.

Example 1 (The word ϵ) The empty word ϵ is represented by the term $f(\perp, \perp)$. See Figure 1a.

Example 2 (The word 1) The word 1 is represented by the term $f(f(\perp, \perp), \perp)$. See Figure 1b.

Example 3 (The word 21221) The word 21221 is represented by the term $f(\perp, f(f(\perp, f(\perp, f(f(\perp, \perp), \perp))), \perp))$. See Figure 1c.

3.1.2 Enforcing a Word Tree

We want to enforce with a first-order formula of subtyping constraints that a tree t is an f -spine, *i.e.*, that it represents a word w . Any f -spine t satisfies three properties:

1. Only f and \perp appear in t (Lemma 3).
2. There is exactly one maximal path of f 's (Lemma 4).
3. t is not \perp (because \perp does not represent a word).

Lemma 3 A tree t contains only f and \perp iff

$$\exists x.((x \leq f(x, x)) \wedge (t \leq x))$$

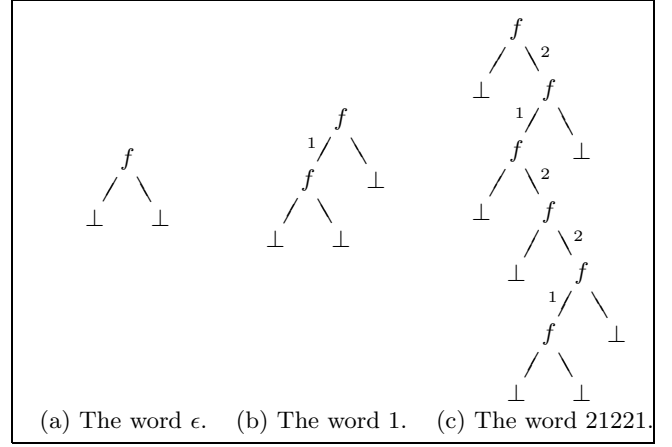


Figure 1: Some example representations of words.

Proof. Suppose t contains only f and \perp . Let h be the height of t , which is the length of the longest branch of t . The full binary tree s of height h where all the leaves are labelled \perp and all the internal nodes are labelled f satisfies $s \leq f(s, s)$ and $t \leq s$.

On the other hand, suppose for some s with $s \leq f(s, s)$, we have $t \leq s$. It suffices to show that s contains only f and \perp . For the sake of argument, assume on some shortest path π from the root, s is labelled with g , *i.e.*, every path strictly shorter than π is labelled either f or \perp . Now consider the path π in $f(s, s)$. If π exists in $f(s, s)$, then it must be labelled either f or \perp in $f(s, s)$. If π does not exist in $f(s, s)$, then a prefix of π exists in $f(s, s)$ and must be labelled with \perp . In both cases, a contradiction is reached since $s \leq f(s, s)$. \square

Lemma 4 For any non- \perp tree t with f and \perp , there is exactly one maximal path of f 's iff the subtypes of t form a chain w.r.t. \leq .

Proof. If t has exactly one maximal path of f 's, then clearly that all the subtypes of t form a chain. On the other hand, if t has at least two maximal paths of f 's. The two subtypes of t where we replace f by \perp at the respective paths are incomparable. \square

Thus we can enforce a tree to represent a word. We shorthand the formula by $\text{word}(t)$, that is

$$\begin{aligned} \text{dom-closure}(t) &\stackrel{\text{def}}{=} \exists x.((x \leq f(x, x)) \wedge (t \leq x)) \\ \text{chain}(t) &\stackrel{\text{def}}{=} \forall t_1, t_2.(((t_1 \leq t) \wedge (t_2 \leq t)) \rightarrow \\ &\quad ((t_1 \leq t_2) \vee (t_2 \leq t_1))) \\ \text{word}(t) &\stackrel{\text{def}}{=} \text{dom-closure}(t) \wedge \text{chain}(t) \wedge (t \neq \perp) \end{aligned}$$

3.1.3 Prepending Trees

In the following discussion, we use words and trees that represent words interchangeably, since the context should make the distinction clear.

To construct a solution to a PCP instance, we need to concatenate words. Thus we want to express with constraints that a word w_1 is obtained from w_2 by prepending w . We express this with a family of predicates prepend_w , one

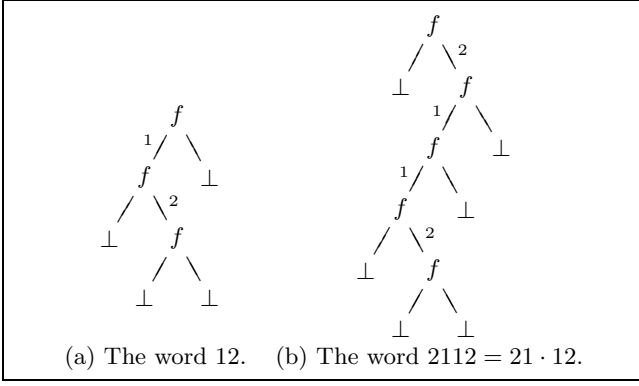


Figure 2: Tree prepending example.

for each constant word w . The predicate $\text{prepend}_w(t_1, t_2)$ is true if the word represented by t_1 is obtained by prepending w to the word for t_2 . Note that this is sufficient, because in PCP, the words are constant. We define the predicate recursively

$$\begin{aligned} \text{prepend}_\epsilon(t_1, t_2) &\stackrel{\text{def}}{=} (t_1 = t_2) \\ \text{prepend}_{1w}(t_1, t_2) &\stackrel{\text{def}}{=} \exists t'. ((t_1 = f(t', \perp)) \wedge \text{prepend}_w(t', t_2)) \\ \text{prepend}_{2w}(t_1, t_2) &\stackrel{\text{def}}{=} \exists t'. ((t_1 = f(\perp, t')) \wedge \text{prepend}_w(t', t_2)) \end{aligned}$$

Example 4 (Prepending example) We prepend the word 21 onto the word 12 (Figure 2a) to get the word 2112 (Figure 2b).

3.2 Reducing PCP to FOT of Subtyping

In this section, we describe how to reduce an instance of PCP to a first-order formula of subtyping constraints.

3.2.1 Outline of the Reduction

We construct a formula that accepts the representations of all the solutions of a PCP instance.

We first describe a solution to a PCP instance as a tree. Recall that a PCP instance P consists of n pairs of words $\langle l_1, r_1 \rangle, \dots, \langle l_n, r_n \rangle$, where $l_i, r_i \in \{1, 2\}^*$. A solution $s = s_1 \dots s_m$ to P is a *non-empty* finite sequence of indices 1 through n , *i.e.*, $s \in \{1, \dots, n\}^+$, such that $l_{s_1} \dots l_{s_m} = r_{s_1} \dots r_{s_m}$. One can represent a solution s as the tree t shown in Figure 3. In the tree t , the values of ϵ , l_{s_1} , r_{s_1} , \dots , $l_{s_m} \dots l_{s_1}$, and $r_{s_m} \dots r_{s_1}$ are represented by their corresponding word trees. The tree is constructed as follows. We start with the empty word pair $\langle \epsilon, \epsilon \rangle$. At each step, we prepend a particular pair from the PCP instance $\langle l_{s_i}, r_{s_i} \rangle$ to the previous pair of words. At the end, $l_{s_1} \dots l_{s_m} = r_{s_1} \dots r_{s_m}$, *i.e.*, we have found a solution to P . Notice that the solutions are constructed in the reverse order because we use prepend instead of append.²

²We use prepend because append is just not as convenient to express.

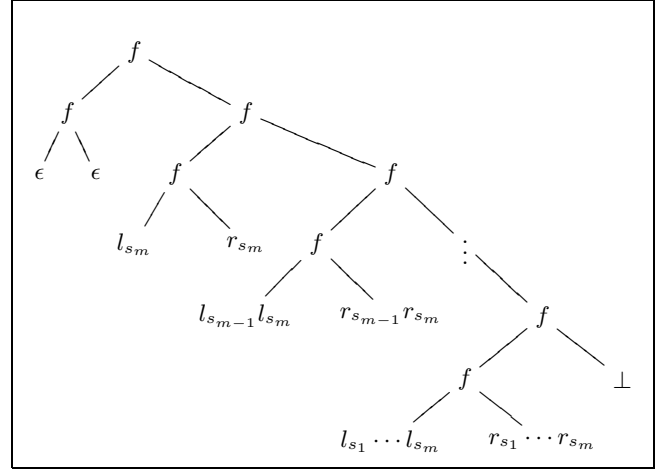


Figure 3: A PCP solution viewed as a tree.

With this representation of PCP solutions as trees, we can reduce an instance of PCP to the validity of a first-order formula of subtyping constraints by expressing that there exists a tree t such that

1. *The tree t is of the particular form in Figure 3.* (Section 3.2.2)

Our construction does not require the branches of the solution tree to be in the order shown in Figure 3. Any order is fine.

2. *We have a valid PCP construction sequence.* (Section 3.2.3)

Each left branch $f(w_i, w'_i)$ is either the pair of empty words or there exists another left branch $f(w_j, w'_j)$ such that $\text{prepend}_{l_k}(w_i, w_j)$ and $\text{prepend}_{r_k}(w'_i, w'_j)$ for some k . In addition, one of the left branches is of the form $f(w, w)$ with w non-empty.³ This ensures that we have a non-empty sequence.

We next express these requirements with first-order formulae of subtyping constraints.

3.2.2 Correct Form of the Tree

To ensure the correct form of the tree t , we require that each left branch represents two words conjoined with the root labelled with f , *i.e.*, we have $f(w, w')$ for some trees representing words w and w' . In order to achieve this, we construct trees of the form shown in Figure 4a, which is a branch of the tree representing a PCP solution shown in Figure 3.

Let t be the tree representing a PCP solution. We cannot extract a branch directly from t because subtyping constraints cannot express removing something from a tree. However, we observe that a branch is a supertype of the *main spine* shown in Figure 4b with some additional properties, which we enforce separately. We first express the main spine s of t . Two properties are needed for s :

³We assume for any PCP instance, $l_i \neq r_i$ for any i . Otherwise, the instance is trivially solvable.

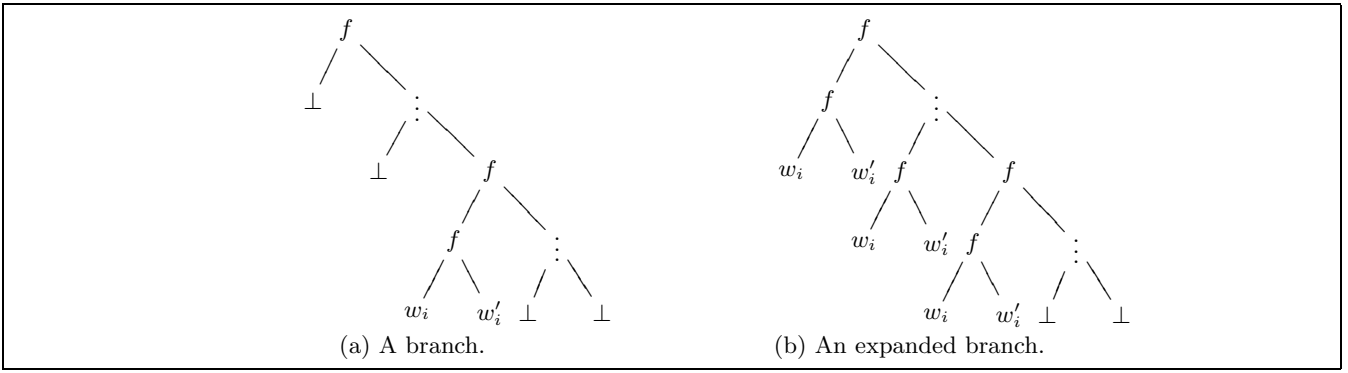


Figure 5: Extracting words from a branch.

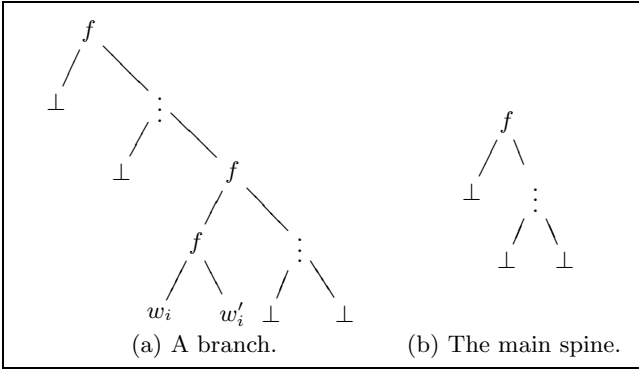


Figure 4: The branch of a solution tree.

1. The main spine s is of the form shown in Figure 4b. We simply require $s \leq f(\perp, s)$.
2. The tree s is a subtype of t and among all possible spines, it is the largest such tree.

This is easily expressed as

$$(s \leq t) \wedge \forall x.(((x \leq f(\perp, x)) \wedge (x \leq t)) \rightarrow (x \leq s))$$

We introduce the shorthand that s is the main spine of t by

$$\text{spine}(s, t) \stackrel{\text{def}}{=} (s \leq f(\perp, s)) \wedge (s \leq t) \wedge (\forall x.(((x \leq f(\perp, x)) \wedge (x \leq t)) \rightarrow (x \leq s)))$$

We observe that a branch b of t is a subtype of t and a proper supertype of the main spine s with two additional properties:

1. Exactly one left branch of the main spine is of the form $f(w_i, w'_i)$.
2. All the other left branches of the main spine are labelled with \perp .

We can express that b is a proper supertype of the main spine s by

$$s < b \stackrel{\text{def}}{=} ((s \leq b) \wedge (s \neq b))$$

We express (1) and (2) by observing that b is a *maximal tree* such that the set of all the subtypes of b that are proper supertypes of the main spine s have a *unique* minimal element, *i.e.*, the set $\{x \mid s < x \leq b\}$ has a unique minimal element. We use $\text{is-min}(u, v, w)$ to express that u is a minimal element of the subtypes of v that are proper supertypes of w , that is

$$\text{is-min}(u, v, w) \stackrel{\text{def}}{=} (u \leq v) \wedge (w < u) \wedge \forall x.(((x \leq v) \wedge (w < x)) \rightarrow (u \leq x))$$

In addition, $\text{uniq-min}(u, w)$ expresses that all the subtypes of u that are proper supertypes w have a *unique* minimal element, that is

$$\text{uniq-min}(u, w) \stackrel{\text{def}}{=} \exists x.(\text{is-min}(x, u, w) \wedge \forall y.(\text{is-min}(y, u, w) \rightarrow (x = y)))$$

With that, we can express the requirements on b by the following formula

$$\text{branch}(b, t) \stackrel{\text{def}}{=} (b \leq t) \wedge \exists s.(\text{spine}(s, t) \wedge (s < b) \wedge \text{uniq-min}(b, s) \wedge \forall x.((b < x \leq t) \rightarrow \neg \text{uniq-min}(x, s)))$$

We establish the correctness of $\text{branch}(b, t)$ in Lemma 5.

Lemma 5 A tree b is a branch of t as shown in Figure 4a iff $\text{branch}(b, t)$.

Proof. It is straightforward to verify that if b is a branch of t then $\text{branch}(b, t)$. For the other direction, assume $\text{branch}(b, t)$. Then we know that b is a subtype of t and a proper supertype of the main spine s . Since $\text{uniq-min}(b, s)$, *i.e.*, all the subtypes of b strictly larger than s have a unique minimum, b cannot have two left sub-branches labelled with f . Thus b must be a subtype of a branch. However, since b is the largest tree such that $\text{uniq-min}(b, s)$, it must be a branch. \square

3.2.3 Correct Construction of the Tree

The previous section describes how to extract a branch of the tree t . However, that is not sufficient, since we ultimately need the two words w_i, w'_i associated with a branch.

We must ensure that for each branch the two words w_i and w'_i are empty or are constructed from the words of another branch w_j and w'_j by prepending l_k and r_k respectively, for some k .

For a branch b , we need to extract the two words w_i and w'_i . The trick is to duplicate the non- \perp left child of b to all the left children of b preceding this non- \perp child. In particular, this would have the effect of duplicating the two words at the first child of the branch.

We give an example. Consider the branch b shown in Figure 5a. We would like to build from b the expanded tree b' shown in Figure 5b. If we can construct such a tree b' , then it is easy to extract the two words w_i and w'_i simply by the constraint $\exists u.f(f(w_i, w'_i), u) = b'$.

We now show how to construct b' from b . Observe that the right child of b' is a subtype of b' itself, *i.e.*, if we let $b' = f(u, v)$, then $v \leq b'$. In addition, observe that of all supertypes of b , b' is the smallest tree with this property. We write the shorthand $\text{recurse}(t_1, t_2)$ for the formula

$$\text{recurse}(t_1, t_2) \stackrel{\text{def}}{=} (t_1 \leq t_2) \wedge \exists x_1, x_2. (t_2 = f(x_1, x_2)) \wedge (x_2 \leq t_2)$$

which says that t_1 is a subtype of t_2 and the right child of t_2 is a subtype of t_2 itself. Now we can express the duplication of b to get b' through the following formula

$$\text{dup-branch}(b, b') \stackrel{\text{def}}{=} \text{recurse}(b, b') \wedge \forall t. (\text{recurse}(b, t) \rightarrow (b' \leq t))$$

We establish the correctness of $\text{dup-branch}(b, b')$ in Lemma 6.

Lemma 6 Let b be a branch of t . A tree b' duplicates the non- \perp sub-branch of b (as shown in Figure 5) iff $\text{dup-branch}(b, b')$.

Proof. It is straightforward to verify that if b' duplicates the non- \perp sub-branch of b , then $\text{dup-branch}(b, b')$. For the other direction, assume $\text{dup-branch}(b, b')$. Since b' (shown in Figure 5b) meets the condition $\text{recurse}(b, b')$, by definition of dup-branch we have $b'' \leq b'$. We also have $b \leq b''$ because $\text{recurse}(b, b'')$ holds. With a simple induction on the height of the left spine of f 's of b , we can show that b'' must be the same as b' . Thus, b'' duplicates the non- \perp sub-branch of b . \square

We introduce a few shorthands next. The formula $\text{wordpair}(w_1, w_2, b, t)$ expresses that for a branch b of a solution tree t , w_1 and w_2 are the pair of words associated with that branch.

$$\text{wordpair}(w_1, w_2, b, t) \stackrel{\text{def}}{=} \text{word}(w_1) \wedge \text{word}(w_2) \wedge \exists b'. (\text{dup-branch}(b, b') \wedge \exists u. (f(f(w_1, w_2), u) = b'))$$

The formula $\text{onestep}(w_i, w'_i, w_j, w'_j)$ expresses a step in the PCP construction, *i.e.*, the concatenation of a pair of words onto the current pair. It says that the words w_i and w'_i are obtained from the words w_j and w'_j by respectively prepending some words l_k and r_k of the PCP instance.

$$\text{onestep}(w_i, w'_i, w_j, w'_j) \stackrel{\text{def}}{=} \bigvee_{1 \leq k \leq n} (\text{prepend}_{l_k}(w_i, w_j) \wedge \text{prepend}_{r_k}(w'_i, w'_j))$$

We can now express that the tree t represents a solution of a PCP instance. Recall that we must express that for each w_i and w'_i , either w_i and w'_i are the empty words, or there exist w_j and w'_j such that $\text{prepend}_{l_k}(w_i, w_j)$ and $\text{prepend}_{r_k}(w'_i, w'_j)$. Consider the PCP instance P in which

we have $\langle l_1, r_1 \rangle, \dots, \langle l_n, r_n \rangle$, where l_i and r_i are words in $\{1, 2\}$. We construct a first-order formula $\text{solvable}(P)$ which is valid iff P is solvable. The formula expresses the existence of a tree representing a solution to P .

We introduce a few more shorthands. The formula $\text{empty}(w)$ tests whether a word w is ϵ . The formula $\text{construct}(w_1, w_2, b', t)$ ensures that w_1 and w_2 are obtained from some branch b' of t by a one step construction. We use $\text{valid-branch}(b, t)$ for saying that the words w_1 and w_2 are either ϵ or are obtained by a construction step of PCP from another branch b' . Finally, we use the formula $\text{accept-branch}(b, t)$ to say that for some branch, the two words associated with that branch are the same and not the empty words ϵ .

$$\begin{aligned} \text{empty}(w) &\stackrel{\text{def}}{=} w = f(\perp, \perp) \\ \text{construct}(w_1, w_2, b', t) &\stackrel{\text{def}}{=} \text{branch}(b', t) \wedge \exists w'_1, w'_2. (\text{wordpair}(w'_1, w'_2, b', t) \wedge \text{onestep}(w_1, w_2, w'_1, w'_2)) \\ \text{valid-branch}(b, t) &\stackrel{\text{def}}{=} (\exists w_1, w_2. \text{wordpair}(w_1, w_2, b, t) \wedge ((\text{empty}(w_1) \wedge \text{empty}(w_2)) \vee \exists b'. \text{construct}(w_1, w_2, b', t))) \\ \text{accept-branch}(b, t) &\stackrel{\text{def}}{=} \text{branch}(b, t) \wedge \exists w. (\text{wordpair}(w, w, b, t) \wedge \neg \text{empty}(w)) \end{aligned}$$

The formula $\text{solvable}(P)$ now can be given as

$$\text{solvable}(P) \stackrel{\text{def}}{=} \exists t. (\forall b. (\text{branch}(b, t) \rightarrow \text{valid-branch}(b, t)) \wedge \exists b. \text{accept-branch}(b, t))$$

The correctness of the reduction from PCP to the first-order theory of subtyping constraints is established in Theorem 7.

Theorem 7 (Soundness and Completeness) A PCP instance P has a solution iff the formula $\text{solvable}(P)$ is valid.

Proof. It is easy to verify that if P has a solution, then any representation of the solution sequence in terms of a tree t shown in Figure 3 meets the requirement

$$\forall b. (\text{branch}(b, t) \rightarrow \text{valid-branch}(b, t)) \wedge \exists b. \text{accept-branch}(b, t)$$

On the other hand, suppose we have such a t , then it is also easy to extract a solution sequence from t . Start with the branch b_m such that the two words associated with b_m are the same. Since b_m is a branch and the two words are not ϵ , there must be another branch b_{m-1} such that we have a PCP construction step. This process must terminate, since t is a finite tree. This reasoning can be easily formalized with an induction on the number of branches of t (or equivalently the size of t). \square

3.3 Recursive Types

In this section, we show that the construction can be adapted to recursive types. Recall that in recursive types, types are interpreted as regular trees over f and \perp .

To adapt our construction, notice that it is sufficient to restrict all the types (trees) to be finite trees. That is, we need only express that a tree t is finite.

It turns out that only the words we get from a branch of t must be finite. The other trees in the construction

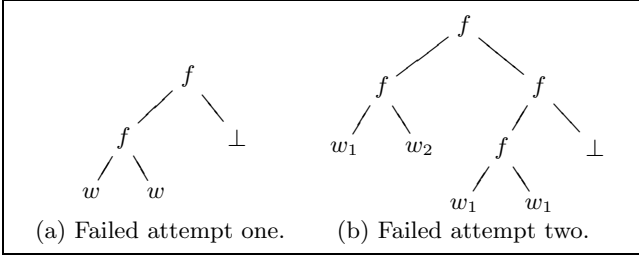


Figure 6: Failed attempts for recursive types.

can be infinite. For words, if we do not restrict them to be finite, the existence of such a tree t as in Figure 3 may not correspond to a solution to the PCP problem. To see this, consider the PCP instance $\{\langle 11, 1 \rangle\}$. Clearly, it has no solution. However, consider the tree $(f(f(w, w), \perp))$ shown in Figure 6a, where w is the infinite regular tree such that $w = f(w, \perp)$, *i.e.*, the infinite word 1^ω .

One may wonder whether we can instead require that a construction step must use two different branches, and that the words for the two branches are not the same at the respective positions. This does not work either. Consider the PCP instance $\{\langle \epsilon, 1 \rangle, \langle \epsilon, 2 \rangle\}$, which has no solution. Now consider the tree $f(f(w_1, w_2), f(f(w_1, w_1), \perp))$ shown in Figure 6b, where $w_1 = f(w_2, \perp) \wedge w_2 = f(\perp, w_1)$, *i.e.*, w_1 is the infinite word $(12)^\omega$ and w_2 is the infinite word $(21)^\omega$.

We take the approach of restricting the words extracted from a branch to be finite. This can be achieved by simply requiring that the set of proper subtypes of w has a *largest element*, *i.e.*,

$$\text{has-max}(w) \stackrel{\text{def}}{=} \exists t.(t < w \wedge \forall t'.(t' < w \rightarrow t' \leq t)$$

Lemma 8 A tree t representing a word is finite iff $\text{has-max}(t)$.

Proof. Let t be a word tree. If t is finite, then the set of proper subtypes of t forms a chain. The set is finite, and thus has a largest element. On the other hand, if the tree is infinite, then all its proper subtypes are finite trees truncated from t , *i.e.*, the set of trees representing the finite prefixes of word denoted by t (except \perp). This set forms an infinite ascending chain, and thus it does not have a largest element. \square

We can now directly use the construction in Section 3, except we require in the formula $\text{wordpair}(w_1, w_2, b, t)$ that w_1 and w_2 are finite:

$$\text{wordpair}(w_1, w_2, b, t) \stackrel{\text{def}}{=} \begin{aligned} & \text{word}(w_1) \wedge \text{word}(w_2) \wedge \\ & \text{has-max}(w_1) \wedge \text{has-max}(w_2) \wedge \\ & \exists b'.(\text{dup-branch}(b, b') \wedge \\ & \exists u.(f(f(w_1, w_2), u) = b')) \end{aligned}$$

Thus, we have shown that the first-order theory of non-structural subtyping constraints over recursive types (and infinite trees) is undecidable.

Theorem 9 The first-order theory of non-structural subtyping constraints over recursive types (and infinite trees) is undecidable for any type language with a binary type symbol and \perp .

Proof. Follows from Lemma 8 and Theorem 7. \square

4 Structural Subtyping: A Comparison

We show that the first-order theory of structural subtyping constraints over the type language over f and \perp is decidable. This result provides a clear contrast between the expressiveness of structural and non-structural subtyping. In addition, it provides another, and in some sense more apparent, distinction between these two alternative interpretation of subtypes. In fact, we show that the first-order theory of structural subtyping constraints with a signature containing one constant symbol is decidable.

Theorem 10 The first-order theory of structural subtyping constraints with a single constant symbol is decidable for both simple and recursive types (and infinite trees).

Proof. This can be easily shown by noticing that in a type language with only one constant (*i.e.*, \perp), the subtype relation is the same as equality. Thus we can simply turn any constraint $t_1 \leq t_2$ into $t_1 = t_2$. Since the first-order theory of equality is decidable both for finite and regular trees (and infinite trees) [21], the theorem follows immediately. \square

5 Decidability of FOT with Unary Symbols

In this section, we show that if we restrict our type language to unary function symbols and constants, the first-order theory is decidable. This result shows that the difficulty in the whole first-order theory lies in binary type constructors. The idea of the proof is to reduce the problem to the tree automata emptiness problem.

Note that word automata would suffice for encoding the case with unary function symbols. However, because our approach is extensible to type languages over arbitrary signatures for the existential or universal fragments (see Section 5.3), we present our results in terms of tree automata.

5.1 Background on Tree Automata

We recall some definitions and results on tree automata.

Tree automata generalize word automata by accepting trees instead of words. Let \mathcal{F} be a ranked alphabet, and let \mathcal{F}_n denote the set of symbols of arity n .

Definition 11 (Finite Tree) A *finite tree* t over a ranked alphabet \mathcal{F} is a mapping from a prefix-closed set $\text{pos}(t) \subseteq \mathbb{N}^*$ into \mathcal{F} . The set of *positions* pos of t satisfies

- $\text{pos}(t)$ is nonempty and prefix-closed.
- For each $\pi \in \text{pos}(t)$, if $t(\pi) \in \mathcal{F}_n$, then $\pi i \in \text{pos}(t)$ for $1 \leq i \leq n$.

Definition 12 (Finite Tree Automata (NFTA)) A *finite tree automaton* (NFTA) over \mathcal{F} is a tuple

$$\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$$

where Q is a finite set of *states*, \mathcal{F} is a finite set of *ranked alphabet*, $Q_F \subseteq Q$ is a set of *final states*, and Δ is a set of *transition rules* of the form

$$f(q_1, \dots, q_n) \longrightarrow q$$

where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \dots, q_n \in Q$.

The above defines a *bottom-up tree automaton*, since an automaton starts at the leaves and works up the tree inductively. The *move relation* of a tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ can be defined as tree rewriting rules $t \xrightarrow{\mathcal{A}} t'$.

We say that $t \xrightarrow{\mathcal{A}} t'$ if t' can be obtained from t by replacing $f(q_1, \dots, q_n)$ with q for some $f(q_1, \dots, q_n) \rightarrow q \in \Delta$. We denote the reflexive and transitive closure of $\xrightarrow{\mathcal{A}}$ by $\xrightarrow{\mathcal{A}^*}$.

A term (or a tree) is *accepted* by a NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ if $t \xrightarrow{\mathcal{A}^*} q$ for some final state q in Q_F .

Example 5 (tree automaton) Consider the automaton where

$$\begin{aligned} Q &= \{q, q_f\} \\ \mathcal{F} &= \{a, b, f(\cdot, \cdot)\} \\ Q_F &= \{q_f\} \\ \Delta &= \left\{ \begin{array}{ll} a & \rightarrow q \\ b & \rightarrow q_f \\ f(q, q_f) & \rightarrow q_f \end{array} \right\} \end{aligned}$$

The automaton accepts the smallest tree language L satisfying (1) $b \in L$, and (2) if $t \in L$ then $f(a, t) \in L$. For example, it accepts the term $f(a, b)$ since

$$\begin{array}{c} f \\ / \quad \backslash \\ a \quad b \end{array} \rightarrow \begin{array}{c} f \\ / \quad \backslash \\ q \quad b \end{array} \rightarrow \begin{array}{c} f \\ / \quad \backslash \\ q \quad q_f \end{array} \rightarrow q_f$$

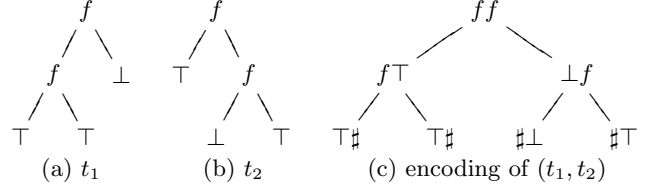
Our goal is to use tree automata to encode the solutions of subtyping constraints. The solutions of a constraint system are an n -ary relation, associating with each type variable a component in the relation. Thus, the solutions of a constraint system of m variables can be represented as a set of m -tuples of trees. For example, the tuple $\langle f(f(\top, \top), \perp), f(\top, f(\perp, \top)) \rangle$ is a solution to the constraint $x \leq y$.

We use a standard encoding to represent tuples [9]. We first give an example to illustrate how the encoding works. Consider tuples of words over the alphabet $\{0, 1\}$. We can construct an automaton to accept the (encoding of) language L of pairs (w, w') such that $\|w\| = \|w'\|$ ($\|w\|$ denotes the length of the word w) and $w_i \neq w'_i$ for $1 \leq i \leq \|w\|$, *i.e.*, we flip 0's and 1's in w and w' . One possible encoding is to “stack” the two words, *i.e.*, put one on top of the other, and we consider the product alphabet $\begin{Bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{Bmatrix}$. With this encoding, we can easily construct an automaton that accepts L , for example, the automaton with one state q and q is both initial and final, having transitions $(q, \begin{smallmatrix} 0 \\ 1 \end{smallmatrix}) \rightarrow q$ and $(q, \begin{smallmatrix} 1 \\ 0 \end{smallmatrix}) \rightarrow q$.

This idea can be extended to tree automata on tuples with “overlapping” of the terms. For any finite ranked alphabet \mathcal{F} , we define $\mathcal{F}^n = (\mathcal{F} \cup \{\#\})^n$, where $\#$ is a new symbol of arity 0. We consider only binary terms, since general n -ary symbols can be simulated with a linear number of binary symbols in the arity of the symbol. We define the arity of the symbols as the maximum of the arities of the components, *i.e.*, $\text{arity}(f_1, \dots, f_n) = \max\{\text{arity}(f_1), \dots, \text{arity}(f_n)\}$. Since $\#$ is of arity 0, the symbol $(\#, \dots, \#)$ is of arity 0, *i.e.*, a constant. We denote by \mathcal{F}_m^n the set of symbols in \mathcal{F}^n of arity m .

For example, consider $\mathcal{F} = \{a, f(\cdot, \cdot)\}$, where a is a constant and f is a binary symbol. Then \mathcal{F}^2 is the set of symbols $\{aa, af, a\#, fa, ff, f\#, \#a, \#f, \#\#\}$ and \mathcal{F}_2^2 is $\{af, fa, ff, f\#, \#f\}$.

Example 6 (tuple encoding)



Definition 13 (Tree Automata on Tuples) Let \mathcal{F} be a ranked alphabet. A *finite tree automaton on n -tuples* over \mathcal{F} is a tree automaton $\mathcal{A} = (Q, \mathcal{F}^n, Q_F, \Delta)$ over \mathcal{F}^n (defined above), where Q is a finite set of *states*, $Q_F \subseteq Q$ is a set of *final states*, and Δ is a set of *transition rules* of the form

$$f(q_1, \dots, q_m) \rightarrow q$$

where $n \geq 0$, $f \in \mathcal{F}_m^n$, $q, q_1, \dots, q_m \in Q$.

Example 7 (automaton on tuples) Consider the automaton where

$$\begin{aligned} Q &= \{q_f\} \\ \mathcal{F} &= \{a, f(\cdot, \cdot)\} \\ Q_F &= \{q_f\} \\ \Delta &= \left\{ \begin{array}{ll} aa & \rightarrow q_f \\ ff(q_f, q_f) & \rightarrow q_f \end{array} \right\} \end{aligned}$$

One can verify that this automaton accepts the tree language $\{(t, t) \mid t \in T(\mathcal{F})\}$.

Let $t = (f_1, \dots, f_i, \dots, f_n)$. Define $t^i = f_i$ (the i -th component of t) and $t^{-i} = (f_1, \dots, f_{i-1}, f_{i+1}, \dots, f_n)$ (the i -th projection of t).

We now define two important operations on relations, projection and cylindrification.

Definition 14 (Projection and Cylindrification) If $R \subseteq T(\mathcal{F})^n$ ($n \geq 1$) and $1 \leq i \leq n$, then the i -th projection of R is the relation $R^i \subseteq T(\mathcal{F})^{n-1}$ defined by

$$R^i(t_1, \dots, t_{n-1}) \Leftrightarrow \exists t \in T(\mathcal{F}). R(t_1, \dots, t_{i-1}, t, t_i, \dots, t_{n-1})$$

If $R \subseteq T(\mathcal{F})^n$ ($n \geq 0$) and $1 \leq i \leq n+1$, then the i -th cylindrification of R is the relation $R^i \subseteq T(\mathcal{F})^{n+1}$ defined by

$$R^i(t_1, \dots, t_{i-1}, t, t_i, \dots, t_n) \Leftrightarrow R(t_1, \dots, t_{i-1}, t_i, \dots, t_n)$$

We summarize here results on tree automata that we use. More details can be found in [9, 15].

Theorem 15 (Decidable Emptiness) The emptiness problem for tree automata is decidable. In fact, it can be decided in linear time in the size of the automaton.

Theorem 16 (Closure Properties) Tree automata are closed under intersection, union, complementation, cylindrification, and projection.

One can view intersection as the equivalent of Boolean “and” \wedge , union as the Boolean “or” \vee , complementation as the Boolean negation \neg , projection as existential quantification \exists . Cylindrification is used to ensure that two automata represent solutions over a common set of variables, so that their intersection can be taken.

5.2 A Decision Procedure

Recall that we consider a monadic signature in this section. We reduce the validity of a formula ϕ to the emptiness decision of a tree automaton. We proceed by structural induction on the formula ϕ . We assume the formula is normalized so that it uses only the connectives \wedge , \neg , and \exists . In addition, w.l.o.g., we assume the literals of the formula are of the form $x \leq y$, $x = \perp$, $x = \top$, and $x = f(y)$.

- $\exists x.\phi$

Let \mathcal{A}_1 be the automaton for ϕ . We construct an automaton \mathcal{A} for $\exists x.\phi$ by taking the projection of \mathcal{A}_1 w.r.t. the x component of the tuple.⁴

- $\neg\phi$

Let \mathcal{A}_1 be the automaton for ϕ . We construct an automaton \mathcal{A} for $\neg\phi$ by complementing \mathcal{A}_1 .

- $\phi_1 \wedge \phi_2$

Let \mathcal{A}_1 and \mathcal{A}_2 be the automata for ϕ_1 and ϕ_2 . We construct \mathcal{A}'_1 and \mathcal{A}'_2 for ϕ_1 and ϕ_2 by cylindrifying \mathcal{A}_1 and \mathcal{A}_2 so that \mathcal{A}'_1 and \mathcal{A}'_2 agree on all the components. Then construct \mathcal{A} for $\phi_1 \wedge \phi_2$ by intersecting \mathcal{A}'_1 and \mathcal{A}'_2 .

The following are for the base predicates.

- $x = \perp$

We construct the automaton

$$\mathcal{A} = (\{q_f\}, \mathcal{F}^1, \{q_f\}, \{\perp \longrightarrow q_f\})$$

- $x = \top$

We construct the automaton

$$\mathcal{A} = (\{q_f\}, \mathcal{F}^1, \{q_f\}, \{\top \longrightarrow q_f\})$$

- $x = f(y)$

We illustrate the construction for the case where there is one other unary function symbol g in addition to f . The constants are \perp and \top .

We construct the following automaton

$$\mathcal{A} = (\{q_f, q_g, q_\perp, q_\top, q_\# \}, \mathcal{F}^2, \{q_f\}, \Delta)$$

to accept all the pairs of (x, y) where $x = f(y)$.

We give a recursive construction of the transitions. We use q_s as the state in which we are expecting a s for the x -component (the first component).

Here are the cases where we expect a f for the x component and in which we accept.

$$\begin{aligned} f\perp(q_\perp) &\longrightarrow q_f \\ f\top(q_\top) &\longrightarrow q_f \\ ff(q_f) &\longrightarrow q_f \\ fg(q_g) &\longrightarrow q_f \end{aligned}$$

Here are the cases where a g is expected for the x component.

$$\begin{aligned} g\perp(q_\perp) &\longrightarrow q_g \\ g\top(q_\top) &\longrightarrow q_g \\ gf(q_f) &\longrightarrow q_g \\ gg(q_g) &\longrightarrow q_g \end{aligned}$$

Here are the base cases.

$$\begin{aligned} \perp\# &\longrightarrow q_\perp \\ \top\# &\longrightarrow q_\top \end{aligned}$$

One can easily show with an induction that the constructed automaton accepts the language $\{(x, y) \mid x = f(y)\}$.

- $x \leq y$

We illustrate the construction for f . We assume f is covariant in its argument. The construction is easily extensible to the case with more function symbols, with function symbols of binary or greater arities, and with function symbols with contravariant arguments.

For $\alpha \leq \beta$ to hold, we have the following cases

- α is \perp ;
- β is \top ;
- $\alpha = f(\alpha_1)$ and $\beta = f(\beta_1)$, where $\alpha_1 \leq \beta_1$.

We construct the automaton

$$\mathcal{A} = (\{q_l, q_r, q_f\}, \mathcal{F}^2, \{q_f\}, \Delta)$$

The transition relation Δ is constructed in pieces.

We have the atomic cases where α and β are either \perp or \top

$$\begin{aligned} \perp\perp &\longrightarrow q_f \\ \perp\top &\longrightarrow q_f \\ \top\top &\longrightarrow q_f \end{aligned}$$

Then we have the cases where $\alpha = \perp$ and $\beta = f(\beta_1)$ or $\beta = \top$ and $\alpha = f(\alpha_1)$.

$$\begin{aligned} \perp f(q_l) &\longrightarrow q_f \\ f\top(q_r) &\longrightarrow q_f \end{aligned}$$

The state q_l is used to signify that the left component can only be $\#$, *i.e.*, the component isn't there. We still need to complete the right component. For q_l , we have the rules

$$\begin{aligned} \#\perp &\longrightarrow q_l \\ \#\top &\longrightarrow q_l \\ \#f(q_l) &\longrightarrow q_l \end{aligned}$$

⁴Notice that only trees that are encodings of tuples of trees are considered during an automata projection.

The case for q_r is symmetric, and we have the rules

$$\begin{aligned} \perp_{\#} &\longrightarrow q_r \\ \top_{\#} &\longrightarrow q_r \\ f_{\#}(q_r) &\longrightarrow q_r \end{aligned}$$

Finally we have the case where $\alpha = f(\alpha_1)$ and $\beta = f(\beta_1)$. In this case, we require the subterms to be related. Thus we have the rule

$$ff(q_f) \longrightarrow q_f$$

One can easily verify that the automaton indeed recognizes the solutions of $\alpha \leq \beta$.

Thus the first-order theory of non-structural subtyping restricted to unary function symbols is decidable. In addition, note that for structural subtyping, the only changes are in the case $x \leq y$, and can be easily expressed with tree automaton. By using an acceptor model for infinite trees and using top-down automata, we can easily adapt this construction for infinite words.

Theorem 17 The first-order theory of non-structural subtyping with unary function symbols is decidable. This holds both for the finite and infinite words and for structural subtyping as well.

Proof. Follows immediately from the above construction and the properties of tree automata. \square

5.3 Extending to Arbitrary Signatures

We now discuss the issues with extending the described approach to arbitrary signatures. There are two related difficulties in extending our approach to the full first-order theory over arbitrary signatures. First, although we can easily express the solutions to $x \leq y$ with standard tree automata, we cannot express the solutions to $x = f(y, z)$ with standard tree automata for any binary symbol f , because the set $\{\langle t_1, t_2, t_3 \rangle \mid t_1 = f(t_2, t_3)\}$ is not a regular set [9]. An extended form of tree automata on tuples is required, *tree automata on tuples with component-wise tests* (TACT); such automata allow machines to test relationships between tuple components [41]. Because this class of tree automata is not closed under projection, it does not extend to the full first-order theory. However, this class of automata is still interesting because it can encode the existential or equivalently the universal fragments of the first-order theory. Therefore, we can reduce non-structural subtype entailment to the emptiness problem on a restricted class of TACT. We believe this reduction is a promising direction in resolving the decidability of non-structural subtype entailment.

6 Related Work

There are a few previous results on constraint simplification regarding subtyping and set constraints. Henglein and Rehof consider the problem of subtyping constraint entailment of the form $C \vDash \alpha \leq \beta$, where C is a constraint set with subtyping constraints and α and β are type variables [17, 18]. The types are constructed from a finite lattice of base elements with the function (\rightarrow) and product (\times) constructors. They prove the following results:

1. Structural subtype entailment over finite types is coNP-complete [17].
2. Structural subtype entailment over recursive types is PSPACE-complete [18].
3. Non-structural subtype entailment over finite types is PSPACE-hard [18].
4. Non-structural subtype entailment over recursive types is PSPACE-hard [18].

Niehren and Priesnitz consider the problem of non-structural subtype entailment. They show that a natural subproblem is PSPACE-complete [27] and characterize non-structural subtype entailment over the signature $\{f(\cdot), \perp, \top\}$ with so-called P-automata [28]. They leave open the decidability of non-structural subtype entailment for this particular signature. Furthermore, it is not known whether this approach can be extended to work on arbitrary signatures.

Niehren *et al.* consider the entailment problem of *atomic set constraints*, a restricted class of set constraints without union and intersections and interpreted over the Herbrand universe. They show entailment of the form $C \vDash \alpha \subseteq \beta$ is PSPACE-complete for atomic set constraints [26]. Flanagan and Felleisen consider the problem of simplifying a variant of atomic set constraints. They show restricted entailment $C_1 \vDash_E C_2$ for this class of constraints is decidable (in exponential time) by reducing restricted entailment to regular tree grammar containment and PSPACE-hard by a reduction from nondeterministic finite state automata containment [13]. They do not give an exact characterization of the complexity of the problem.

Entailment problems for conditional equality constraints [38] (a weaker form of non-structural subtyping constraints) are studied in [39]. Both entailment and restricted entailment are PTIME-complete. This is in contrast to a simple extension, for which entailment is still PTIME-complete, but restricted entailment is coNP-complete.

A few researchers consider semantic notions for subtyping constraint simplification. The most powerful one is the notion of *observational equivalence* defined in [42]. Intuitively, observational equivalence says that from the analysis point of view replacing one constraint set with an equivalent one does not change the observable behavior of the constraint system. A similar notion is used in [33] for simplifying subtyping constraints.

There is also related work in term rewriting and constraint solving over trees in general [8, 10]. The work in this paper is inspired by work in this area. Maher shows the first-order theory of finite trees, infinite trees, and rational trees is decidable by giving a complete axiomatization [21]. Many researchers consider various order relations among trees, similar to the subtype orders. Venkataraman study the first-order theory of subterm ordering over finite trees. The existential fragment is shown to be NP-complete and the $\exists\forall$ -fragment to be undecidable [43]. Müller *et al.* study the first order theory of feature trees and show it undecidable [25]. Comon and Treinen show the first-order theory of lexicographic path ordering is undecidable [11]. Automata-theoretic constructions are used to obtain decidability results for many theories. Büchi uses finite word automata to show the decidability of WS1S and S1S [7]. Finite automata are also used to construct alternative proofs of decidability of Presburger arithmetic [6, 44], and Rabin's decidability of WS2S and S2S are based on tree automata [35].

7 Conclusion

In this paper, we have shown that the first-order theory of non-structural subtyping constraints is undecidable via a reduction from the Post's Correspondence Problem (PCP). The result holds both for finite and infinite trees and for any type signature with at least one binary type constructor and a least element \perp . This result yields a technical separation of structural subtyping and non-structural subtyping. We have also shown that the first-order theory of subtyping constraints with unary function symbols is decidable. The automata-theoretic construction bridges automata theory and subtyping problems. This provides us with an alternative way of tackling of the problems.

We consider this work a step towards resolving the long-standing open questions about subtyping. The most outstanding problems are the decidability of non-structural subtyping entailment and subtyping constrained types.

References

- [1] 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, Copenhagen, Denmark, June 1993.
- [2] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft Typing with Conditional Types. In *Proceedings of the 21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, January 1994.
- [3] A. Aiken, E. Wimmers, and J. Palsberg. Optimal Representations of Polymorphic Types with Subtyping. In *Proceedings of 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, pages 47–76, 1997.
- [4] R. Amadio and L. Cardelli. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
- [5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.
- [6] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.
- [7] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.
- [8] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.
- [9] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, 1999.
- [10] H. Comon and R. Treinen. Ordering constraints on trees. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, 11-13 April 1994. Springer Verlag.
- [11] H. Comon and R. Treinen. The first-order theory of lexicographic path orderings is undecidable. *Theoretical Computer Science*, 176:67–87, April 1997.
- [12] M. Fähndrich and A. Aiken. Making Set-Constraint Based Program Analyses Scale. In *First Workshop on Set Constraints at CP'96*, Cambridge, MA, August 1996. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.
- [13] C. Flanagan and M. Felleisen. Componential Set-Based Analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.
- [14] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.
- [15] F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
- [16] N. Heintze. Set Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.
- [17] F. Henglein and J. Rehof. The Complexity of Subtype Entailment for Simple Types. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 352–361, 1997.
- [18] F. Henglein and J. Rehof. Constraint Automata and the Complexity of Recursive Subtype Entailment. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 616–627, 1998.
- [19] D. Kozen, J. Palsberg, and M.I. Schwartzbach. Efficient Recursive Subtyping. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 419–428, 1993.
- [20] D. Kozen, J. Palsberg, and M.I. Schwartzbach. Efficient Inference of Partial Types. *Journal of Computer and System Sciences (JCSS)*, 49(2):306–324, 1994.
- [21] M.J. Maher. Complete axiomatizations of the algebras of the finite, rational and infinite trees. In *Proceedings of the Third IEEE Symposium on Logic in Computer Science*, pages 348–357, Edinburgh, UK, 1988. Computer Society Press.
- [22] S. Marlow and P. Wadler. A Practical Subtyping System For Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*, pages 136–149, June 1997.
- [23] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

- [24] J.C. Mitchell. Type Inference with Simple Types. *Journal of Functional Programming*, 1(3):245–285, 1991.
- [25] M. Müller, J. Niehren, and R. Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science*, 4(2):193–234, September 2001.
- [26] J. Niehren, M. Müller, and J. Talbot. Entailment of Atomic Set Constraints is PSPACE-Complete. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 285–294, 1999.
- [27] J. Niehren and T. Priesnitz. Entailment of Non-Structural Subtype Constraints. In *Asian Computing Science Conference*, number 1742 in LNCS, pages 251–265. Springer, December 1999.
- [28] J. Niehren and T. Priesnitz. Non-Structural Subtype Entailment in Automata Theory. In *Proceedings of 4th International Symposium on Theoretical Aspects of Computer Software (TACS’01)*, number 2215 in LNCS, pages 360–384. Springer, 2001.
- [29] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.
- [30] J. Palsberg and P. O’Keefe. A Type System Equivalent to Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995.
- [31] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the ACM Conference on Object-Oriented programming: Systems, Languages, and Applications*, pages 146–161, October 1991.
- [32] E.L. Post. A Variant of a Recursively Unsolvable Problem. *Bull. of the Am. Math. Soc.*, 52, 1946.
- [33] F. Pottier. Simplifying Subtyping Constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP ’96)*, pages 122–133, May 1996.
- [34] F. Pottier. Simplifying subtyping constraints: a theory. To appear in *Information & Computation*, August 2000.
- [35] M.O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.
- [36] J. Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, 1998.
- [37] O. Shivers. Control Flow Analysis in Scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.
- [38] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.
- [39] Z. Su and A. Aiken. Entailment with conditional equality constraints. In *Proceedings of European Symposium on Programming*, pages 170–189, April 2001.
- [40] R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–457, November 1992.
- [41] R. Treinen. Predicate logic and tree automata with tests. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures*, volume LNCS 1784, pages 329–343. Springer, 2000.
- [42] V. Trifonov and S. Smith. Subtyping Constrained Types. In *Proceedings of the 3rd International Static Analysis Symposium*, pages 349–365, September 1996.
- [43] K.N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM*, 34(2):492–510, April 1987.
- [44] P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints. In *Static Analysis Symposium*, pages 21–32, 1995.

A An Example

We give an example in this section to demonstrate our automata construction in Section 5. Consider the alphabet $\mathcal{F} = \{\perp, \top, g(\cdot)\}$. We want to decide the entailment $\{x \leq g(y), g(x) \leq y\} \models x \leq y$.

This entailment holds. We reason with a proof by contradiction. Suppose the entailment does not hold. Then there exist two trees t_1 and t_2 such that (1) $t_1 \leq g(t_2)$ and $g(t_1) \leq t_2$; and (2) $t_1 \not\leq t_2$. Choose t_1 and t_2 to be trees such that $\|t_1\| + \|t_2\|$ is minimized. Notice that $t_1 = g(t'_1)$ and $t_2 = g(t'_2)$ for some t'_1 and t'_2 , otherwise, t_1 and t_2 cannot witness the non-entailment. However, then we have $g(t'_1) \leq g(g(t'_2))$, i.e., $t'_1 \leq g(t'_2)$ and $g(g(t'_1)) \leq g(t'_2)$, i.e., $g(t'_1) \leq t'_2$. Furthermore, $t'_1 \not\leq t'_2$ since $t_1 = g(t'_1) \not\leq g(t'_2) = t_2$. Thus, t'_1 and t'_2 also witness the non-entailment, a contradiction.

We demonstrate that the entailment holds with the technique presented in this paper. After flattening the constraints, we consider the equivalent entailment

$$\{x' = g(x), y' = g(y), x \leq y', x' \leq y\} \models x \leq y$$

The above entailment is equivalent to deciding whether the constraints $\{x' = g(x), y' = g(y), x \leq y', x' \leq y, x \not\leq y\}$ are unsatisfiable.

We construct an automaton for each of the five constraints.

- $x' = g(x)$

Consider the automaton where

$$\begin{aligned} Q &= \{q_1, q_2, q_f\} \\ \mathcal{F} &= \{\perp, \top, g(\cdot)\}^2 \\ Q_F &= \{q_f\} \\ \Delta &= \left\{ \begin{array}{ll} \#\perp & \longrightarrow q_1 \\ \perp g(q_1) & \longrightarrow q_f \\ \#\top & \longrightarrow q_2 \\ \top g(q_2) & \longrightarrow q_f \\ gg(q_f) & \longrightarrow q_f \end{array} \right\} \end{aligned}$$

The first component is for x , and the second component is for x' .

- $y' = g(y)$

This is the same automaton as for $x' = g(x)$, with the first component for y and the second component for y' .

- $x \leq y'$

Consider the automaton where

$$\begin{aligned} Q &= \{q_1, q_2, q_f\} \\ \mathcal{F} &= \{\perp, \top, g(\cdot)\}^2 \\ Q_F &= \{q_f\} \\ \Delta &= \left\{ \begin{array}{l} \perp\perp \longrightarrow q_f \\ \perp\top \longrightarrow q_f \\ \top\top \longrightarrow q_f \\ \# \perp \longrightarrow q_1 \\ \# \top \longrightarrow q_1 \\ \# g(q_1) \longrightarrow q_1 \\ \perp g(q_1) \longrightarrow q_f \\ \perp \# \longrightarrow q_2 \\ \top \# \longrightarrow q_2 \\ g \# (q_2) \longrightarrow q_2 \\ g \top (q_2) \longrightarrow q_f \\ gg(q_f) \longrightarrow q_f \end{array} \right\} \end{aligned}$$

The first component is for x , and the second component is for y' .

- $x' \leq y$

This is the same automaton as for $x \leq y'$, with the first component for x' and the second component for y .

- $x \not\leq y$

Consider the automaton where

$$\begin{aligned} Q &= \{q_1, q_2, q_f\} \\ \mathcal{F} &= \{\perp, \top, g(\cdot)\}^2 \\ Q_F &= \{q_f\} \\ \Delta &= \left\{ \begin{array}{l} \top\perp \longrightarrow q_f \\ \# \perp \longrightarrow q_1 \\ \# \top \longrightarrow q_1 \\ \# g(q_1) \longrightarrow q_1 \\ \top g(q_1) \longrightarrow q_f \\ \perp \# \longrightarrow q_2 \\ \top \# \longrightarrow q_2 \\ g \# (q_2) \longrightarrow q_2 \\ g \perp (q_2) \longrightarrow q_f \\ gg(q_f) \longrightarrow q_f \end{array} \right\} \end{aligned}$$

The first component is for x , and the second component is for y .

Now we apply cylindrification to the automata above.⁵ We use the following shorthand for transition rules:

$$f_1(f_2 \mid f_3)(q) \longrightarrow q'$$

⁵Before applying cylindrification, we need to make these automata complete. Because of space limitations and the tediousness of the construction, we simply use the original automata to illustrate how cylindrification works. The basic construction is the same regardless whether the automata are complete or not.

is a shorthand for the two rules

$$f_1 f_2(q) \longrightarrow q'$$

and

$$f_1 f_3(q) \longrightarrow q'$$

For $x' = g(x)$, consider the automaton where

$$\begin{aligned} Q &= \{q_1, q_2, q_f\} \\ \mathcal{F} &= \{\perp, \top, g(\cdot)\}^4 \\ Q_F &= \{q_f\} \end{aligned}$$

$$\Delta = \left\{ \begin{array}{l} /* derived from \# \perp \longrightarrow q_1 */ \\ \#(\# \mid \perp \mid \top) \perp(\# \mid \perp \mid \top) \longrightarrow q_1 \\ \# g \perp(\# \mid \perp \mid \top) g(q_1) \longrightarrow q_1 \\ \# g \perp(\# \mid \perp \mid \top) g(q_2) \longrightarrow q_1 \\ \# g \perp(\# \mid \perp \mid \top) g(q_f) \longrightarrow q_1 \\ \#(\# \mid \perp \mid \top) \perp g(q_1) \longrightarrow q_1 \\ \#(\# \mid \perp \mid \top) \perp g(q_2) \longrightarrow q_1 \\ \#(\# \mid \perp \mid \top) \perp g(q_f) \longrightarrow q_1 \\ /* derived from \perp g(q_1) \longrightarrow q_f */ \\ \perp g(q_1) \longrightarrow q_f \\ \perp(\# \mid \perp \mid \top) g(\# \mid \perp \mid \top) g(q_1) \longrightarrow q_f \\ /* derived from \# \top \longrightarrow q_2 */ \\ \#(\# \mid \perp \mid \top) \top(\# \mid \perp \mid \top) \longrightarrow q_2 \\ \# g \top(\# \mid \perp \mid \top) g(q_1) \longrightarrow q_2 \\ \# g \top(\# \mid \perp \mid \top) g(q_2) \longrightarrow q_2 \\ \# g \top(\# \mid \perp \mid \top) g(q_f) \longrightarrow q_2 \\ \#(\# \mid \perp \mid \top) \top g(q_1) \longrightarrow q_2 \\ \#(\# \mid \perp \mid \top) \top g(q_2) \longrightarrow q_2 \\ \#(\# \mid \perp \mid \top) \top g(q_f) \longrightarrow q_2 \\ /* derived from \top g(q_2) \longrightarrow q_f */ \\ \top(\# \mid \perp \mid \top) g(\# \mid \perp \mid \top) g(q_2) \longrightarrow q_f \\ /* derived from gg(q_f) \longrightarrow q_f */ \\ g(\# \mid \perp \mid \top) g(\# \mid \perp \mid \top) g(q_f) \longrightarrow q_f \end{array} \right\}$$

This automaton is obtained from the automaton for $x' = g(x)$ above by applying cylindrification twice. The tuples are ordered by x, y, x' , and y' , *i.e.*, the first component corresponds to x , the second component corresponds to y , and so on.

The other four cases are treated in exactly the same manner.

- $y' = g(y)$
- $x \leq y'$
- $x' \leq y$
- $x \not\leq y$

Then we can construct the intersection of the five automata obtained through cylindrification and verify that the language accepted by the intersection is empty. With that, we conclude that the entailment does indeed hold. Due to space limitations, the rest of the construction is left to the reader.