The First-Order Theory of Subtype Constraints

ZHENDONG SU
University of California, Davis
and
ALEXANDER AIKEN
Stanford University
and
JOACHIM NIEHREN
INRIA Futurs, Lille
and
TIM PRIESNITZ
Universität des Saarlandes
and
RALF TREINEN
ENS de Cachan, CNRS and INRIA Futurs

We investigate the first-order theory of subtype 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. Our results hold for both simple and recursive types. The undecidability result is shown by a reduction from the Post's Correspondence Problem, and the decidability result is shown by a reduction to a decision problem on tree automata. In addition, we introduce the notion of a constrained tree automaton to express non-structural subtype entailment. This work is a step towards resolving long-standing open problems of the decidability of entailment for non-structural subtyping.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics,syntax*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*constraints,polymorphism*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*; F.4.0 [**Mathematical Logic and Formal Languages**]: Formal Languages; F.4.0 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: subtype constraints, type systems, tree automata, complexity

---

An earlier version [Su et al. 2002] of the paper was published in the Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 203–216, January 2002.
Authors' addresses: Zhendong Su, Department of Computer Science, University of California, Davis, CA 95616-8562. Alexander Aiken: Computer Science Department, Stanford University, Stanford, CA 94305-9025. Joachim Niehren: Mostrare Project, INRIA Futurs, Lille, France. Tim Priesnitz: Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany. Ralf Treinen: LSV, CNRS UMR 8643 and INRIA Futurs, ENS de Cachan, 94235 Cachan Cedex, France.

1. INTRODUCTION

In this paper we present the first decidability and undecidability results for the first-order theory of subtyping.[1] 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 [Mitchell 1991], 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, [Aiken et al. 1994; Andersen 1994; Flanagan et al. 1996; Heintze 1994; Marlow and Wadler 1997; Palsberg and Schwartzbach 1991; Shivers 1988]) and, more speculatively, richer designs for typed languages ([Odersky and Wadler 1997]).

Subtyping algorithms invariably involve systems of subtype constraints $\tau_1 \leq \tau_2$, where the τ_i are types that may contain type variables. There are two interesting questions we can ask about a system of subtype 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 subtype constraints are by now quite well understood (e.g., see [Rehof 1998]). 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 model 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 [Odersky and Wadler 1997]).

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 $\bot$ and a greatest type $\top$, so that $\bot \leq \tau \leq \top$ for any τ. More details on structural and non-structural subtyping can be found in [Mitchell 1991; Amadio and Cardelli 1993; Kozen et al. 1993].

Despite extensive effort over many years, the exact complexity and even the decidability of entailment is open for non-structural subtype constraints [Rehof 1998; Henglein and Rehof 1997; 1998; Flanagan and Felleisen 1997; Niehren and Priesnitz 1999; 2003; Trifonov and Smith 1996; Aiken et al. 1997; Fähndrich and Aiken 1996; Pottier 1996; Marlow and Wadler 1997; Pottier 2001]. As we show in Section 2, the natural versions of entailment

[1] This paper extends [Su et al. 2002] with a notion of constrained tree automata to express non-structural subtype entailment and an undecidability result of the emptiness problem over *general* constrained tree automata (cf. Section 5.3).

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 subtype constraints is undecidable via a reduction from the Post's Correspondence Problem (PCP). The result is shown for both simple and recursive types (Theorems 3.8 and 3.10). The result holds also for infinite trees. In addition, this result yields a technical separation of structural subtyping and non-structural subtyping (Theorem 4.1).

—We show that the first-order theory of subtype 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 5.10).

—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).

It was shown recently that the first-order theory of structural subtyping over simple types is decidable [Kuncak and Rinard 2003]. The decidability of the full first-order theory of structural subtyping over recursive types is open.

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 subtype constraints and show that the first-order theory of subtype 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. SUBTYPE 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 [Milner 1978]. We consider the following type language

$$\tau ::= \bot \mid \top \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$$

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

Types in this language form a lattice with the following ordering:

—$\bot \leq \tau \leq \top$, for any τ;

—$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ 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.*, $\bot \leq \bot \to \top$. 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 $\alpha = \alpha \to \bot$. 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

$$\{\bot, \top, \cdot \to \cdot, \cdot \times \cdot\}$$

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 subtype 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 [Palsberg and O'Keefe 1995; Pottier 1996; Kozen et al. 1994].

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 [Aiken and Wimmers 1993; Trifonov and Smith 1996; Aiken et al. 1997]. An ML style polymorphic type can be viewed as a constrained type with no constraints. For example,

$$\alpha \to \beta \backslash \{\alpha \leq \textbf{int} \to \textbf{int}, \textbf{int} \to \alpha \leq \beta\}$$

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

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 [Pottier 1996; Marlow and Wadler 1997; Fähndrich and Aiken 1996] 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.[2] 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 [Trifonov and Smith 1996]. Let $\tau_1 \backslash C_1$ and $\tau_2 \backslash C_2$ be two constrained types. We wish to check whether $\tau_1 \backslash C_1 \leq \tau_2 \backslash C_2$ which holds if for every instance τ of $\tau_2 \backslash C_2$, there exists an instance of τ' of $\tau_1 \backslash C_1$ 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

$$\tau_1 \backslash C_1 \leq \tau_2 \backslash C_2 \quad \text{iff} \quad \alpha \backslash (C_1 \cup \{\alpha = \tau_1\}) \leq \beta \backslash (C_2 \cup \{\beta = \tau_2\})$$

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

Although extensive research has directed at these problems [Rehof 1998; Henglein and Rehof 1997; 1998; Flanagan and Felleisen 1997; Niehren and Priesnitz 1999; 2003; Trifonov and Smith 1996; Aiken et al. 1997; Fähndrich and Aiken 1996; Pottier 1996; Marlow and Wadler 1997; Pottier 2001], their decidability has been open for many years. In this paper, we present results on the first-order theory of subtype constraints, which we believe is a step in resolving these open problems.

2.2 The First-Order Theory of Subtype Constraints

We first define the *first-order theory of subtype 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 $Q_1 \ldots Q_n.f$ 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 and the entailment $C \vDash x \leq y$ holds iff the universal formula $\forall x_1, \ldots, x_n.(C \rightarrow (x \leq y))$ is valid, where the x_i's are the variables free in $C \cup \{x \leq y\}$.

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

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:

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

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

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

2.2.3 *Subtype Constrained Types is in the* $\forall\exists$*-Fragment.* Let $\alpha\backslash C_1$ and $\beta\backslash C_2$ be constrained types. We express $\alpha\backslash C_1 \leq \beta\backslash C_2$ as the formula

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

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

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

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

Proposition 2.1. Subtype constrained types is polynomially reducible to existential entailment.

PROOF. We have the following equivalences:

$$\begin{aligned}
& \alpha\backslash C_1 \leq \beta\backslash C_2 \\
\Leftrightarrow \quad & \{ \text{ by defn. of } \alpha\backslash C_1 \leq \beta\backslash C_2 \ \} \\
& S(C_2)|_{\{\beta\}} \subseteq \ S(\alpha \leq \beta \wedge C_1)|_{\{\beta\}} \\
\Leftrightarrow \quad & \{ \text{ by defn. of existential entailment with } E = \text{fv}(C_1) \ \} \\
& C_2 \vDash \exists E.(\alpha \leq \beta \wedge C_1)
\end{aligned}$$

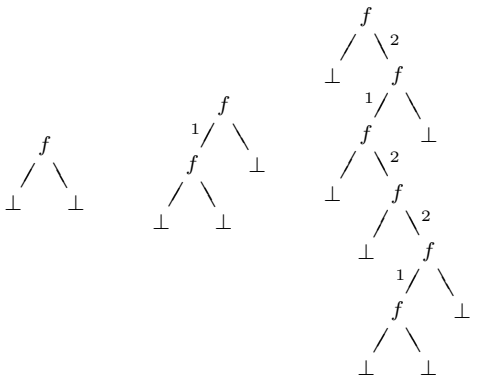
□

3. UNDECIDABILITY OF THE FIRST-ORDER THEORY OF NON-STRUCTURAL SUBTYPING

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 $\exists\forall\exists\forall\exists\forall$-fragment.

The proof is via a reduction from the Post's Correspondence Problem (PCP) [Post 1946] to a first-order formula of non-structural subtyping. Since PCP is undecidable [Post 1946], the first-order theory of non-structural subtyping is undecidable as well. The proof follows the framework of Treinen [Treinen 1992] and is inspired by the proof of undecidability of the first-order theory of ordering constraints over feature trees [Müller et al. 2001].

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) The word ϵ. (b) The word 1. (c) The word 21221.

Fig. 1. Some example representations of words.

a non-empty finite sequence of indices $s_1 \ldots s_m$ (where $1 \leq s_i \leq n$ for $1 \leq i \leq m$) and the sequence constitutes a pair of matched words:

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

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 $\bot$ 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:

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

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.

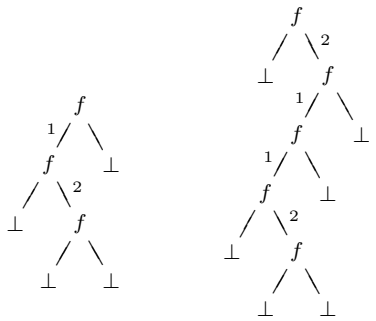
3.1 Representing Words as Trees

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 $\bot$. 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 $\bot$.

Definition 3.1 (f-SPINE). A finite tree t (in f and $\bot$) 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 3.2. The empty word ϵ is represented by the term $f(\bot, \bot)$. See Figure 1a. The word 1 is represented by the term $f(f(\bot, \bot), \bot)$. See Figure 1b. The word 21221 is represented by $f(\bot, f(f(\bot, f(\bot, f(f(\bot, \bot), \bot))), \bot))$. See Figure 1c.



(a) The word 12. (b) The word $2112 = 21 \cdot 12$.

Fig. 2. Tree prepending example.

3.1.2 *Enforcing a Word Tree.* We want to enforce with a first-order formula of subtype 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 $\bot$ appear in t (Lemma 3.3).
(2) There is exactly one maximal path of f's (Lemma 3.4).
(3) t is not $\bot$ (because $\bot$ does not represent a word).

Lemma 3.3. A tree t contains only f and $\bot$ iff $\exists x.((x \leq f(x,x)) \ \wedge \ (t \leq x))$ holds.

PROOF. Suppose t contains only f and $\bot$. 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 $\bot$ 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 $\bot$. 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 $\bot$. Now consider the path π in $f(s,s)$. If π exists in $f(s,s)$, then it must be labelled either f or $\bot$ 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 $\bot$. In both cases, a contradiction is reached since $s \leq f(s,s)$. □

Lemma 3.4. For any non-$\bot$ tree t with f and $\bot$, 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 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 $\bot$ at the respective paths are incomparable. □

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

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

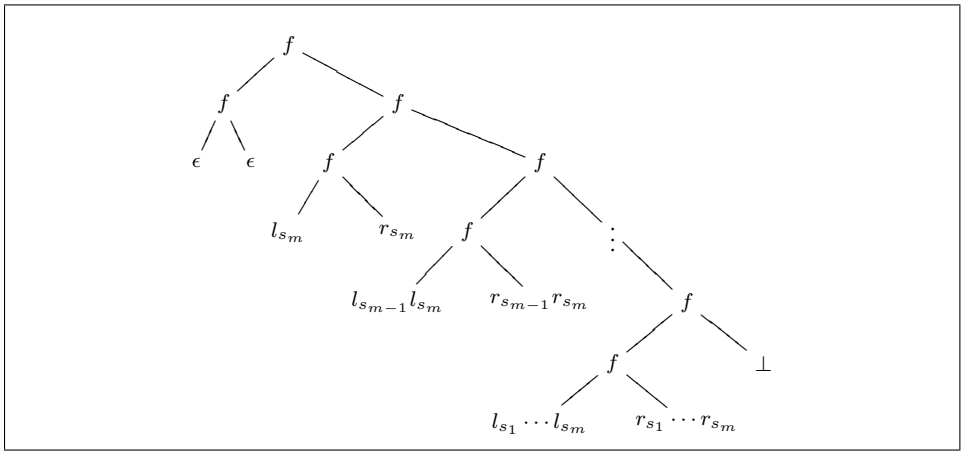


Fig. 3. A PCP solution viewed as a tree.

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 $\mathsf{prepend}_w$, one for each constant word w. The predicate $\mathsf{prepend}_w(t_1, t_2)$ 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:

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

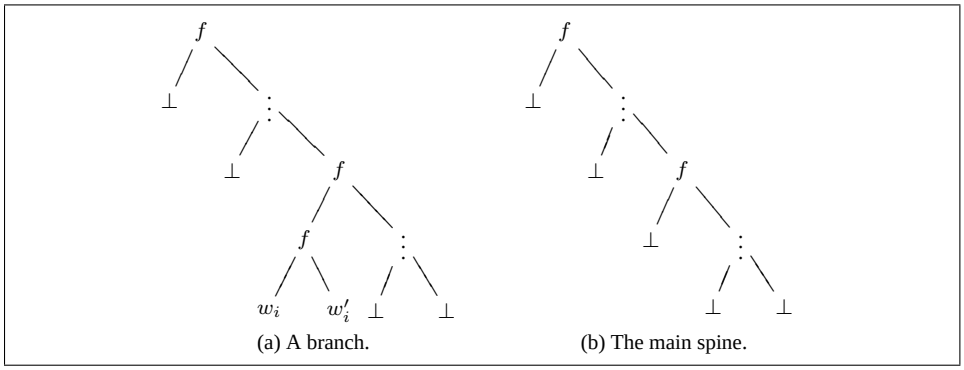
Example 3.5 (PREPENDING). We prepend the word 21 onto the word 12 (Figure 2a) to get the word 2112 (Figure 2b).

3.2 Reducing PCP to the First-Order Theory of Non-structural Subtyping

In this section, we describe how to reduce an instance of PCP to a first-order formula of subtype 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 $\langle l_1, r_1\rangle, \ldots, \langle l_n, r_n\rangle$, where $l_i, r_i \in \{1, 2\}^*$. A solution $s = s_1 \cdots s_m$ to P is a *non-empty* finite sequence of indices 1 through n, *i.e.*, $s \in \{1, \cdots, n\}^+$, such that $l_{s_1} \cdots l_{s_m} = r_{s_1} \cdots r_{s_m}$. One can represent a solution s as the tree t shown in Figure 3. In the tree t, the values of ϵ, l_{s_m}, r_{s_m}, ..., $l_{s_1} \cdots l_{s_m}$, and $r_{s_1} \cdots r_{s_m}$ 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, $l_{s_1} \cdots l_{s_m} = r_{s_1} \cdots r_{s_m}$, *i.e.*,



(a) A branch. (b) The main spine.

Fig. 4. The branch of a solution tree.

we have found a solution to P. Notice that the solutions are constructed in the reverse order because we use prepend instead of append.[3]

With this representation of PCP solutions as trees, we can reduce an instance of PCP to the validity of a first-order formula of subtype 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 (repetitions are also permitted).

(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 $\mathsf{prepend}_{l_k}(w_i, w_j)$ and $\mathsf{prepend}_{r_k}(w'_i, w'_j)$ for some k. In addition, one of the left branches is of the form $f(w, w)$ with w non-empty.[4] This ensures that we have a non-empty sequence.

We next express these requirements with first-order formulae of subtype 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 subtype 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:

(1) *The main spine s is of the form shown in Figure 4b.*
We simply require $s \leq f(\bot, s)$.

(2) *The tree s is a subtype of t and among all possible spines, it is the largest such tree.*

[3] We use prepend because append is just not as convenient to express.
[4] We assume for any PCP instance, $l_i \neq r_i$ for any i. Otherwise, the instance is trivially solvable.

This is easily expressed as:

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

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

$$\begin{array}{rl} \mathsf{spine}(s,t) \stackrel{\text{def}}{=} & (s \leq f(\bot, s)) \ \wedge \ (s \leq t) \\ & (\forall x.(((x \leq f(\bot, x)) \wedge (x \leq t)) \ \rightarrow \ (x \leq s))) \end{array}$$

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* $\bot$.

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 $\mathsf{is\text{-}min}(u, v, w)$ to express that u is the smallest element of the subtypes of v that are proper supertypes of w, that is:

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

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

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

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

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

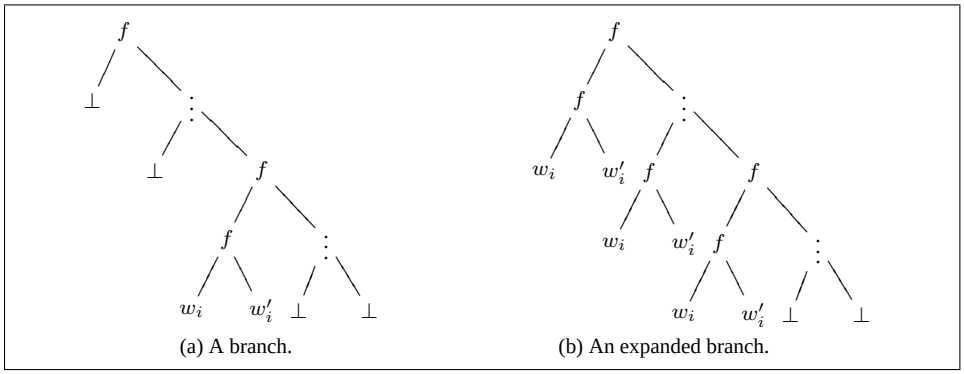
We establish the correctness of $\mathsf{branch}(b, t)$ in Lemma 3.6.

Lemma 3.6. A tree b is a branch of t as shown in Figure 4a iff $\mathsf{branch}(b, t)$.

PROOF. It is straightforward to verify that if b is a branch of t then $\mathsf{branch}(b, t)$. For the other direction, assume $\mathsf{branch}(b, t)$. Then we know that b is a subtype of t and a proper supertype of the main spine s. Since $\mathsf{uniq\text{-}min}(b, s)$, *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 $\mathsf{uniq\text{-}min}(b, s)$, it must be a branch. □

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.



(a) A branch. (b) An expanded branch.

Fig. 5. Extracting words from a branch.

For a branch b, we need to extract the two words w_i and w_i'. The trick is to duplicate the non-$\bot$ left child of b to all the left children of b preceding this non-$\bot$ 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 $\mathsf{recurse}(t_1, t_2)$ for the formula:

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

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:

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

We establish the correctness of $\mathsf{dup\text{-}branch}(b, b')$ in Lemma 3.7.

Lemma 3.7. Let b be a branch of t. A tree b' duplicates the non-$\bot$ sub-branch of b (as shown in Figure 5) iff $\mathsf{dup\text{-}branch}(b, b')$.

PROOF. It is straightforward to verify that if b' duplicates the non-$\bot$ sub-branch of b, then $\mathsf{dup\text{-}branch}(b, b')$. For the other direction, assume $\mathsf{dup\text{-}branch}(b, b'')$. Since b' (shown in Figure 5b) meets the condition $\mathsf{recurse}(b, b')$, by definition of $\mathsf{dup\text{-}branch}$ we have $b'' \leq b'$. We also have $b \leq b''$ because $\mathsf{recurse}(b, b'')$ 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-$\bot$ sub-branch of b. □

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

$$\mathsf{wordpair}(w_1, w_2, b) \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{word}(w_1) \wedge \mathsf{word}(w_2) \wedge \\ \exists b'.(\mathsf{dup\text{-}branch}(b, b') \wedge \exists u.(f(f(w_1, w_2), u) = b')) \end{array}$$

The formula $\mathsf{onestep}(w_i, w_i', w_j, w_j')$ 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.

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

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 $\mathsf{prepend}_{l_k}(w_i, w_j)$ and $\mathsf{prepend}_{r_k}(w_i', w_j')$. Consider the PCP instance P in which we have $\langle l_1, r_1\rangle, \ldots, \langle l_n, r_n\rangle$, where l_i and r_i are words in $\{1, 2\}$. We construct a first-order formula $\mathsf{solvable}(P)$ 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 $\mathsf{empty}(w)$ tests whether a word w is ϵ. The formula $\mathsf{construct}(w_1, w_2, b', t)$ ensures that w_1 and w_2 are obtained from some branch b' of t by a one step construction. We use $\mathsf{valid\text{-}branch}(b, t)$ 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 $\mathsf{accept\text{-}branch}(b, t)$ to say that for some branch, the two words associated with that branch are the same and not the empty words ϵ.

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

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

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

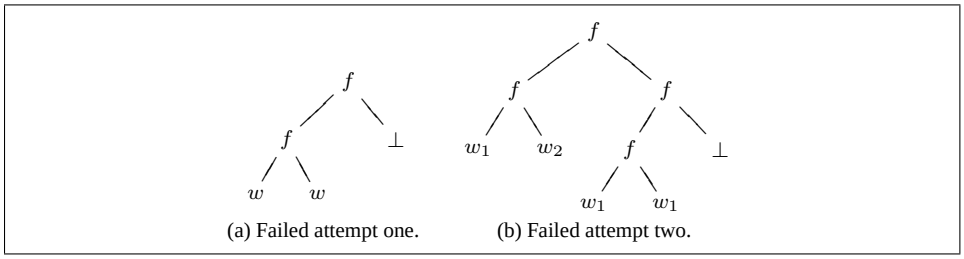
The correctness of the reduction from PCP to the first-order theory of subtype constraints is established in Theorem 3.8.

Theorem 3.8 (SOUNDNESS AND COMPLETENESS). A PCP instance P has a solution iff the formula $\mathsf{solvable}(P)$ 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

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

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). □



(a) Failed attempt one. (b) Failed attempt two.

Fig. 6. Failed attempts for recursive types.

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 $\bot$.

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 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 ($f(f(w, w), \bot)$ shown in Figure 6a, where w is the infinite regular tree such that $w = f(w, \bot)$, *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 $f(f(w_1, w_2), f(f(w_1, w_1), \bot))$ shown in Figure 6b, where $w_1 = f(w_2, \bot) \ \wedge \ w_2 = f(\bot, w_1)$, *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.*,

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

Lemma 3.9. A tree t representing a word is finite iff $\mathsf{has\text{-}max}(t)$.

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 $\bot$). This set forms an infinite ascending chain, and thus it does not have a largest element. □

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

$$\begin{aligned}\mathsf{wordpair}(w_1, w_2, b) \stackrel{\text{def}}{=} \ & \mathsf{word}(w_1) \ \wedge \ \mathsf{word}(w_2) \ \wedge \ \mathsf{has\text{-}max}(w_1) \ \wedge \ \mathsf{has\text{-}max}(w_2) \ \wedge \\ & \exists b'.(\mathsf{dup\text{-}branch}(b, b') \ \wedge \ \exists u.(f(f(w_1, w_2), u) = b'))\end{aligned}$$

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

Theorem 3.10. The first-order theory of non-structural subtype constraints over recursive types (and infinite trees) is undecidable for any type language with a binary type symbol and $\perp$.

PROOF. Follows from Lemma 3.9 and Theorem 3.8. □

4. STRUCTURAL SUBTYPING: A COMPARISON

We show that the first-order theory of structural subtype 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 subtype constraints with a signature containing one constant symbol is decidable.

Theorem 4.1. The first-order theory of structural subtype 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) [Maher 1988], the theorem follows immediately. □

Recently, it is shown that the full first-order theory of finite structural subtyping over arbitrary signatures is decidable [Kuncak and Rinard 2003]. This result, together with our undecidability result, provides a further distinction of structural subtyping from non-structural subtyping. However, at this point, an interesting open problem is the decidability of the full first-order theory of structural subtyping over recursive types.

5. DECIDABILITY OF THE FIRST-ORDER THEORY OF SUBTYPING OVER 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 5.1 (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 $\pi i \in \text{pos}(t)$ iff $1 \leq i \leq n$.

Definition 5.2 (FINITE TREE AUTOMATA). 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

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

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

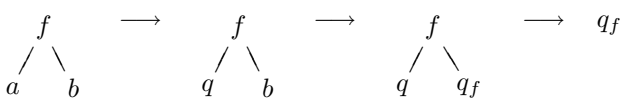
The above defines a *bottom-up tree automaton*, since an automaton starts at the leaves and works up the tree inductively. The *move relation* of a tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ can be defined as tree rewriting rules $t \underset{\mathcal{A}}{\longrightarrow} t'$. We say that $t \underset{\mathcal{A}}{\longrightarrow} t'$ if t' can be obtained from t by replacing $f(q_1, \ldots, q_n)$ with q for some $f(q_1, \ldots, q_n) \longrightarrow q \in \Delta$. We denote the reflexive and transitive closure of $\underset{\mathcal{A}}{\longrightarrow}$ by $\overset{*}{\underset{\mathcal{A}}{\longrightarrow}}$.

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

Example 5.3 (TREE AUTOMATON). Consider the automaton where

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

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



Our goal is to use tree automata to encode the solutions of subtype 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 $\langle f(f(\top, \top), \bot), f(\top, f(\bot, \top))\rangle$ is a solution to the constraint $x \leq y$.

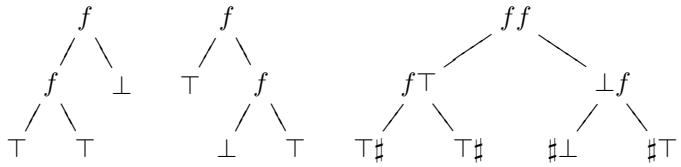
We use a standard encoding to represent tuples [Comon et al. 2002]. We first give an example to illustrate how the encoding works. Consider tuples of words over the alphabet $\{0, 1\}$. We can construct an automaton to accept the (encoding of) language L of pairs (w, w') such that $\|w\| = \|w'\|$ ($\|w\|$ 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 $\{\begin{smallmatrix}0\\0\end{smallmatrix}, \begin{smallmatrix}0\\1\end{smallmatrix}, \begin{smallmatrix}1\\0\end{smallmatrix}, \begin{smallmatrix}1\\1\end{smallmatrix}\}$. 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 $(q, \begin{smallmatrix}0\\1\end{smallmatrix}) \longrightarrow q$ and $(q, \begin{smallmatrix}1\\0\end{smallmatrix}) \longrightarrow q$.

This idea can be extended to tree automata on tuples with "overlapping" of the terms. For any finite ranked alphabet $\mathcal{F}$, we define $\mathcal{F}^n = (\mathcal{F} \cup \{\sharp\})^n$, where $\sharp$ 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.*, $\text{arity}(f_1, \ldots, f_n) = \max\{\text{arity}(f_1), \ldots, \text{arity}(f_n)\}$. Since $\sharp$ is of arity 0, the symbol $(\sharp, \ldots, \sharp)$ is of arity 0, *i.e.*, a constant. We denote by $\mathcal{F}^n_m$ 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 $\{aa, af, a\sharp, fa, ff, f\sharp, \sharp a, \sharp f, \sharp\sharp\}$ and $\mathcal{F}^2_2$ is $\{af, fa, ff, f\sharp, \sharp f\}$.

Example 5.4 (TUPLE ENCODING). Consider the terms $t_1 = f(f(\top, \top), \bot)$ and $t_2 = f(\top, f(\bot, \top))$. We show below how to encode the tuple $\langle t_1, t_2\rangle$.



(a) t_1 (b) t_2 (c) encoding of (t_1, t_2)

Definition 5.5 (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

$$f(q_1, \ldots, q_m) \longrightarrow q$$

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

Example 5.6 AUTOMATON ON TUPLES. Consider the automaton where

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

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

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

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

Definition 5.7 (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

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

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

$$\forall t, t_1, \ldots, t_n \in T(\mathcal{F}).(R^{+i}(t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_n) \Leftrightarrow R(t_1, \ldots, t_{i-1}, t_i, \ldots, t_n))$$

We summarize here results on tree automata that we use. More details can be found in [Gécseg and Steinby 1984; Comon et al. 2002].

Theorem 5.8 (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 5.9 (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 for the Monadic Fragment

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 $x \leq y$, $x = \bot$, $x = \top$, and $x = f(y)$. We first consider the cases with quantifiers and boolean connectives:

$\boxed{\exists x.\phi}$ 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.[5]

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

$\boxed{\phi_1 \wedge \phi_2}$ 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 other cases are for the base predicates:

$\boxed{x = \bot}$ We construct the automaton

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

$\boxed{x = \top}$ We construct the automaton

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

$\boxed{x = f(y)}$ We illustrate the construction for the case where there is one other unary function symbol g in addition to f. The constants are $\bot$ and $\top$. We construct the following automaton

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

[5] Notice that only trees that are encodings of tuples of trees are considered during an automata projection.

to accept all the pairs of (x, y) where $x = f(y)$. The transitions Δ is constructed recursively. We use q_s as the state in which we are expecting an s for the x-component (*i.e.*, the first component).
Here are the cases where we expect an f for the x component and in which we accept:

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

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

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

Here are the base cases:

$$\begin{aligned} \bot\sharp &\longrightarrow q_\bot \\ \top\sharp &\longrightarrow q_\top \end{aligned}$$

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

$\boxed{x \leq y}$ 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 $\bot$;
—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 first have the atomic cases where α and β are either $\bot$ or $\top$:

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

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

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

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

$$\begin{aligned} \sharp\bot &\longrightarrow q_l \\ \sharp\top &\longrightarrow q_l \\ \sharp f(q_l) &\longrightarrow q_l \end{aligned}$$

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

$$\begin{aligned} \bot\sharp &\longrightarrow q_r \\ \top\sharp &\longrightarrow q_r \\ f\sharp(q_r) &\longrightarrow q_r \end{aligned}$$

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:

$$f f(q_f) \longrightarrow q_f$$

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 automata. By using an acceptor model for infinite trees and using top-down automata, we can easily adapt this construction for infinite words.

Theorem 5.10. 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. □

To illustrate our construction, an example encoding of an entailment problem is given in Appendix A.

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 a standard tree automaton, we cannot express the solutions to $x = f(y, z)$ with a standard tree automaton 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 regular [Comon et al. 2002]. An extended form of tree automata on tuples is required, which belongs to the class of *tree automata on tuples with component-wise tests* (TACT). Such automata allow machines to test relationships between tuple components [Treinen 2000]. 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 over a restricted class of TACT. We believe this reduction is a promising direction in resolving the decidability of non-structural subtype entailment.

5.3.1 *Expressing $x \leq y$ over Arbitrary Signatures.* We first show how to encode the solutions to $x \leq y$ for the general case over arbitrary signatures. In particular, it is sufficient to consider the case with a binary function symbol, say f. For $x \leq y$ to hold, we have the following cases:

—x is $\bot$;
—y is $\top$; or
—$x = f(x_1, x_2)$ and $y = f(y_1, y_2)$, and $x_1 \leq y_1$ and $x_2 \leq y_2$.

We construct the automaton

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

with the transition relation Δ constructed in pieces. First, we have the atomic cases where x and y are either $\bot$ or $\top$:

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

Then we have the cases where $x = \bot$ and y is a product type, or $y = \top$ and x is a product type:

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

Similar to our construction for the monadic fragment, the state q_l is used to signify that the left component can only be $\sharp$, *i.e.*, the particular component is not there. We still need to complete the right component. For q_l, we have the rules:

$$\begin{aligned} \sharp\bot &\longrightarrow q_l \\ \sharp\top &\longrightarrow q_l \\ \sharp f(q_l, q_l) &\longrightarrow q_l \end{aligned}$$

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

$$\begin{aligned} \bot\sharp &\longrightarrow q_r \\ \top\sharp &\longrightarrow q_r \\ f\sharp(q_r, q_r) &\longrightarrow q_r \end{aligned}$$

Finally we have the case where both x and y are product types. In this case, we require the corresponding subterms to be related by the subtype ordering. Thus we have the rule:

$$ff(q_f, q_f) \longrightarrow q_f$$

It expresses the following structural decomposition rule:

$$f(\tau_1, \tau_2) \leq f(\tau_1', \tau_2') \text{ iff } \tau_1 \leq \tau_1' \text{ and } \tau_2 \leq \tau_2'$$

One can easily verify that the automaton indeed recognizes the solutions to $x \leq y$.

Because of the closure properties of tree automata, we can obtain an automaton representation of $x \not\leq y$ by complementing the one constructed for $x \leq y$. However, in order to give an additional example, we show a direct construction for $x \not\leq y$. The automaton

construction is similar to the construction for $x \leq y$. The reader is invited to verify the following construction of the automaton

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

where Δ is given by the following transitions:

$$\begin{aligned}
\top\bot &\longrightarrow q_f \\
\top f(q_l, q_l) &\longrightarrow q_f \\
f\bot(q_r, q_r) &\longrightarrow q_f \\
ff(q_f, q) &\longrightarrow q_f \\
ff(q, q_f) &\longrightarrow q_f \\
\\
\sharp\bot &\longrightarrow q_l \\
\sharp\top &\longrightarrow q_l \\
\sharp f(q_l, q_l) &\longrightarrow q_l \\
\\
\bot\sharp &\longrightarrow q_r \\
\top\sharp &\longrightarrow q_r \\
f\sharp(q_r, q_r) &\longrightarrow q_r \\
\\
\bot\bot &\longrightarrow q \\
\bot\top &\longrightarrow q \\
\bot f(q_l, q_l) &\longrightarrow q \\
ff(q, q) &\longrightarrow q \\
f\bot(q_r, q_r) &\longrightarrow q \\
f\top(q_r, q_r) &\longrightarrow q \\
\top\bot &\longrightarrow q \\
\top f(q_l, q_l) &\longrightarrow q \\
\top\top &\longrightarrow q
\end{aligned}$$

The meaning of states q_l and q_r is the same as in the case $x \leq y$. The state q is used to recognize all the possible pairs of terms. Notice that our construction can be easily extended to handle contravariant type constructors.

5.3.2 *Constrained Automata.* Recall that the only type of literals not expressible with standard tree automata is of the form $x = f(y, z)$. Our idea is to separate the regular part of subtype constraints (*i.e.*, expressible with tree automata) and the non-regular part (*i.e.*, not expressible with tree automata). To achieve this, we introduce the notion of a *constrained tree automaton*, a special case of a tree automaton with component-wise tests [Treinen 2000] where tests can only be performed at the root of the tree being tested for acceptance. Interested readers are referred to [Treinen 2000; Comon et al. 2002] for more details on tree automata with tests.

Consider a tree automaton $\mathcal{A}$ on n-tuples over the ranked alphabet $\mathcal{F}$. We name its n components $x_1, \ldots, x_n$. We are interested in the following decision problem: Given $\mathcal{A}$

and C, where C is a set of equations (*i.e.,* unification constraints) over $x_1, \ldots, x_n$ and $\mathcal{F}$, decide whether there exist trees $t_1, \ldots, t_n$ such that:

(1) $\langle t_1, \ldots, t_n \rangle$ is accepted by $\mathcal{A}$.
(2) The valuation h with $h(x_i) = t_i$ satisfies C.

We call such an automaton with equations a *constrained automaton*, and denote it by $\langle \mathcal{A}, C \rangle$. Next, we give an example constrained automaton.

Example 5.11 (CONSTRAINED AUTOMATA). Consider the automaton $\mathcal{A}$ where

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

and the set of equations $C = \{x_1 = f(x_2, x_2)\}$.

Notice that the constrained automaton $\langle \mathcal{A}, C \rangle$ does not accept any finite trees. However, it does accept some infinite trees. For example, take both x_1 and x_2 to be the complete infinite tree, *i.e.,* $x_1 = x_2 = t$, where t is the unique solution to $t = f(t, t)$.

5.3.3 *Expressing Subtype Entailment with Constrained Automata.* We now show how to reduce subtype entailment (structural or non-structural, finite or recursive) to the emptiness problem for constrained automata.

Consider a subtype entailment problem $C \vDash x \leq y$. It holds if and only if the constraint

$$C' \stackrel{\text{def}}{=} C \wedge x \not\leq y$$

does not have a solution, because each solution to C' corresponds to a witness to the non-entailment of $C \vDash x \leq y$. The idea is to use a constrained automaton to express all the solutions to the constraint $C \wedge x \not\leq y$. Let $C = \{\tau_1 \leq \tau_1', \ldots, \tau_n \leq \tau_n'\}$. Then C is equivalent to the single constraint $\tau \leq \tau'$, where $\tau = f(\tau_1, f(\tau_2, f(\ldots, \tau_n)))$ and $\tau' = f(\tau_1', f(\tau_2', f(\ldots, \tau_n')))$. For example, let C be the following set of constraints:

$$\{x_1 \leq f(y_1, f(z_1, \bot)), f(\top, y_1) \leq y_2, f(\bot, z_1) \leq y_1\}$$

It is equivalent to the following constraint:

$$f(x_1, f(f(\top, y_1), f(\bot, z_1))) \leq f(f(y_1, f(z_1, \bot)), f(y_2, y_1))$$

Next, we introduce two fresh type variables x' and y'. Let $\mathcal{A}$ be the tree automaton constructed for the constraints $x' \leq y'$ and $x \not\leq y$. Now, consider the constrained automaton

$$\langle \mathcal{A}, \{x' = \tau, y' = \tau'\} \rangle$$

It is obvious that $\langle \mathcal{A}, \{x' = \tau, y' = \tau'\} \rangle$ is empty if and only if $C \vDash x \leq y$.

This is a simple and straightforward reduction. There is a special property about the constructed automaton for subtype entailment: The tree automata component consists of an automaton with a bounded number of states. Next, we show that general constrained automata emptiness is undecidable. The proof crucially relies upon the fact that the tree automaton component has an unbounded number of states. Therefore, it is open whether constrained automata emptiness is decidable if the associated tree automaton has a bounded

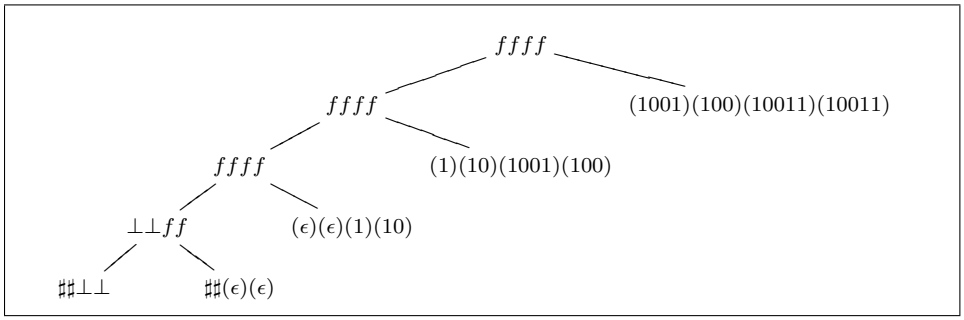


Fig. 7. A solution to the PCP instance $\langle 1, 10\rangle, \langle 001, 0\rangle, \langle 1, 11\rangle$ viewed as a tuple tree.

number of states. Its decidability would imply the decidability of non-structural subtype entailment.

5.3.4 *Undecidability of the Constrained Automata Emptiness Problem.* In this part, we show that the emptiness problem for constrained automata is undecidable. Our result holds for any signature with at least a binary (or larger arity) function symbol $f(,)$ and $\bot$. This again is the smallest signature for our result to hold. Indeed, one can show that emptiness is decidable when only unary symbols and constants are allowed, which follows from the decidability of the monadic fragment (cf. Section 5). Our proof of undecidability of emptiness for constrained automata is through a reduction from PCP.

Recall the PCP problem. Let $\langle p_i, q_i\rangle$ for $1 \leq i \leq n$ be a PCP instance with p_i and q_i being words over 0 and 1. The problem is to decide whether there exists a non-empty sequence $s_1, \ldots, s_m$ with $s_i \in [1, n]$, such that $p_{s_1} \ldots p_{s_m} = q_{s_1} \ldots q_{s_m}$.

We adapt and simplify a proof of Treinen [Treinen 2001]. First, we recall from Section 3.1.1 the encoding of a word w over 0 and 1 with a tree $c(w)$ over the alphabet $\{f(,), \bot\}$:

—$c(\epsilon) = f(\bot, \bot)$;
—$c(0w) = f(c(w), \bot)$;
—$c(1w) = f(\bot, c(w))$.

A key step in our reduction is the encoding of solutions to a PCP instance as 4-tuples of trees over the alphabet $\{f(,), \bot\}$. Certainly there are many possible encodings of PCP solutions, however, we need one that can be recognized by a constrained automaton. We explain our encoding with an example. Consider the PCP instance $\langle 1, 10\rangle, \langle 001, 0\rangle, \langle 1, 11\rangle$. One solution for it is $1.001.1 = 10.0.11 = 10011$. We show how to encode this solution with the tree in Figure 7. In our encoding, for a simpler presentation we use (w) to denote the word w's corresponding tree representation $c(w)$.

We now explain how to interpret our encoding in Figure 7 of the PCP solution. The three subtrees $(\epsilon)(\epsilon)(1)(10)$, $(1)(10)(1001)(100)$, and $(1001)(100)(10011)(10011)$ form a sequence, which corresponds to the three steps in the PCP solution:

—$(\epsilon)(\epsilon)(1)(10)$: appending $\langle 1, 10\rangle$ to $\langle \epsilon, \epsilon\rangle$;
—$(1)(10)(1001)(100)$: appending $\langle 001, 0\rangle$ to $\langle 1, 10\rangle$;
—$(1001)(100)(10011)(10011)$: appending $\langle 1, 11\rangle$ to $\langle 1001, 100\rangle$.

To make sure that our encoding does correspond to a PCP solution, we also need to make sure that the intermediate results are carried over from one step to the next, *i.e.*, the third and fourth components (underlined below) of a step must be the same as the first and second components (underlined below) of the following step:

—From $(\epsilon)(\epsilon)\underline{(1)(10)}$ to $\underline{(1)(10)}(1001)(100)$;
—From $(1)(10)\underline{(1001)(100)}$ to $\underline{(1001)(100)}(10011)(10011)$.

Finally, because the third and fourth components of $(1001)(100)(10011)(10011)$ are the same, we indeed have a solution for the particular PCP instance.

Next, we show how to construct a constrained automaton to recognize all (and only) tree encodings t of solutions to a PCP problem. We proceed with the following steps:

(**Step 1**) To enforce that each right subtree of the main spine of t corresponds to a valid PCP construction step (with a standard tree automaton);
(**Step 2**) To enforce that the right-most subtree's third and fourth components are the same (with a standard tree automaton);
(**Step 3**) To enforce that the correct intermediate results are carried over from one step to the next (with equality constraints).

Step 1. We construct an automaton $\mathcal{A}$ to accept all tree encodings of word tuples of the form $\langle w, w', wp_i, w'q_i\rangle$ for some i, *i.e.*,

$$L(\mathcal{A}) = \{\langle w, w', v, v'\rangle \mid \exists i.v = wp_i \wedge v' = w'q_i\}$$

This automaton is used to encode a single step in the PCP construction. For a particular i, we construct two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ to accept respectively all the tuples $\langle w, wp_i\rangle$ and $\langle w', w'q_i\rangle$. We can then construct $\mathcal{A}$ for the above defined tree language from $\mathcal{A}_1$ and $\mathcal{A}_2$ because tree automata are closed under cylindrification, union, and intersection.

In particular, for the automaton $\mathcal{A}_1$ associated with $p_i = l_1 \ldots l_k$, where $l_{j\in\{1..k\}} \in \{0, 1\}$, we have the following set of transition rules (when $k > 0$):

$$\begin{array}{rcll}
ff(r_f, r_\perp) & \longrightarrow & r_f & \\
ff(r_\perp, r_f) & \longrightarrow & r_f & \\
ff(r_0, r_\perp) & \longrightarrow & r_f & \text{if } l_1 = 0 \\
ff(r_\perp, r_0) & \longrightarrow & r_f & \text{if } l_1 = 1 \\
\perp f(r_{\sharp\perp}, r_{\sharp\perp}) & \longrightarrow & r_0 & \text{if } k = 1 \\
\perp f(r_1, r_{\sharp\perp}) & \longrightarrow & r_0 & \text{if } k > 1 \text{ and } l_2 = 0 \\
\perp f(r_{\sharp\perp}, r_1) & \longrightarrow & r_0 & \text{if } k > 1 \text{ and } l_2 = 1 \\
\sharp f(r_j, r_{\sharp\perp}) & \longrightarrow & r_{j-1} & \text{if } j > 1 \text{ and } l_j = 0 \\
\sharp f(r_{\sharp\perp}, r_j) & \longrightarrow & r_{j-1} & \text{if } j > 1 \text{ and } l_j = 1 \\
\perp\perp & \longrightarrow & r_\perp & \\
\sharp\perp & \longrightarrow & r_{\sharp\perp} & \\
\sharp\perp & \longrightarrow & r_k &
\end{array}$$

For $k = 0$ (*i.e.*, $p_i = \epsilon$), we simply have:

$$\begin{array}{rcl}
ff(r_f, r_\perp) & \longrightarrow & r_f \\
ff(r_\perp, r_f) & \longrightarrow & r_f \\
ff(r_\perp, r_\perp) & \longrightarrow & r_f \\
\perp\perp & \longrightarrow & r_\perp
\end{array}$$

In our construction, the only final state is r_f.

We perform a similar construction for the automaton $\mathcal{A}_2$ that is associated with the word q_i. Then we can apply cylindrification and intersection to obtain an automaton that constructs a PCP step with the particular pair $\langle p_i, q_i \rangle$. We apply the same construction to each i to obtain n automata. With these automata as sub-automata, we can construct an automaton $\mathcal{A}$ to accept the language that represents a PCP step, *i.e.*, $\langle w, w', wp_i, w'q_i \rangle$ for some i, which is achieved by taking the union of the tree automata for each i.

Step 2. In the next step, we construct an automaton to accept all the tuples $\langle w_1, w_2, w, w \rangle$, with the same third and fourth components. We first construct the following automaton to accept all the pairs of equal words $\langle w, w \rangle$:

$$\begin{aligned} \bot\bot &\longrightarrow s_f \\ ff(s_f, s_f) &\longrightarrow s_f \end{aligned}$$

where s_f is the only state and also the final state. By applying cylindrification twice on this automaton, we get an automaton that accepts the language $\langle w_1, w_2, w, w \rangle$. Taking the intersection of this automaton with the automaton $\mathcal{A}$ constructed in the previous step, we get $\mathcal{A}'$.

Step 3. In this final step, in order to ensure that our encoding does correspond to a PCP solution, we need to make sure the intermediate results are carried over from one step to the next correctly. The main difficulty is to enforce this requirement with constraints at the root of the tree. In fact, it is surprising this can be achieved at all. To see what we need to enforce, consider again the example in Figure 7. Let x,y,u,and v corresponds to the first, second, third, and fourth components respectively. Notice that $u = f(x, c(10011))$ and $v = f(y, c(10011))$, where $c(10011)$ denotes the tree encoding of the word 10011. On the other hand, notice that in general, if we require $u = f(x, w)$ and $v = f(y, w)$ for some w, then x,y, u, and v does meet our third requirement.

Here is how our construction works. We construct an automaton with $\mathcal{A}$ and $\mathcal{A}'$ as its sub-automata. Here are the transition rules:

$$\begin{aligned} \sharp\sharp\bot\bot &\longrightarrow t_{\sharp\bot} && \\ \sharp\sharp ff(t_{\sharp\bot}, t_{\sharp\bot}) &\longrightarrow t_{\sharp f} && \\ \bot\bot ff(t_{\sharp\bot}, t_{\sharp f}) &\longrightarrow t_0 && \\ ffff(t_0, r_f) &\longrightarrow t_0 && \text{for every final state } r_f \text{ of } \mathcal{A} \\ ffff(t_0, s_f) &\longrightarrow t && \text{for every final state } s_f \text{ of } \mathcal{A}' \end{aligned}$$

where t is the only accepting state. We then cylindrify this automaton to add a fifth component to obtain our final automaton $\mathcal{A}_P$. It accepts our encoding in Figure 7. The tree where nodes are labelled with their corresponding states is given in Figure 8.

We name the five components of $\mathcal{A}_P$ with the variables x,y,u,v, and w. The equality constraint component of our constrained automaton consists of two equations: $u = f(x, w)$ and $v = f(y, w)$. This step completes our reduction from PCP to the constrained automata emptiness problem. One can verify the correctness of our construction, *i.e.*, the PCP instance P is solvable if and only if $L(\mathcal{A}_P)$ is not empty.

Theorem 5.12 (SOUNDNESS AND COMPLETENESS). The PCP instance P is solvable iff there is a tuple of trees $\langle x, y, u, v, w \rangle \in L(\mathcal{A}_P)$ and satisfies the unification constraints $u = f(x, w)$ and $v = f(y, w)$.

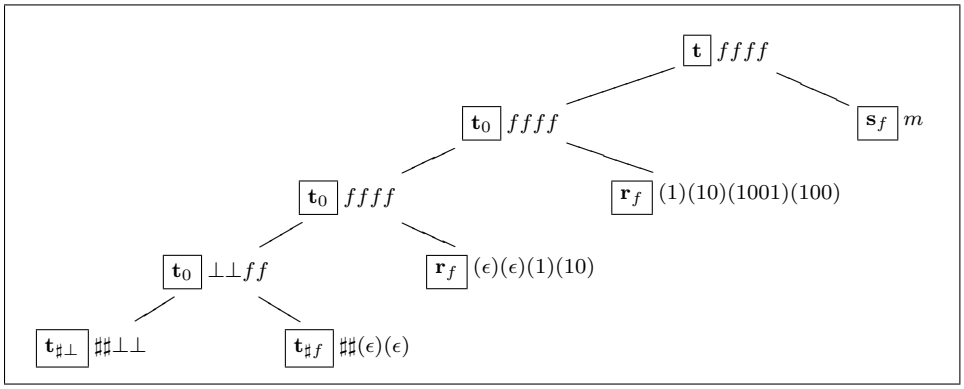


Fig. 8. The solution tree in Figure 7 labelled with state information, where m stands for the word tree $(1001)(100)(10011)(10011)$.

6. RELATED WORK

There are a few previous results on constraint simplification regarding subtype and set constraints. Henglein and Rehof consider the problem of subtype constraint entailment of the form $C \vDash \alpha \leq \beta$, where C is a constraint set with subtype constraints and α and β are type variables [Henglein and Rehof 1997; 1998]. 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 [Henglein and Rehof 1997].

(2) Structural subtype entailment over recursive types is PSPACE-complete [Henglein and Rehof 1998].

(3) Non-structural subtype entailment over finite types is PSPACE-hard [Henglein and Rehof 1998].

(4) Non-structural subtype entailment over recursive types is PSPACE-hard [Henglein and Rehof 1998].

Niehren and Priesnitz consider the problem of non-structural subtype entailment. They show that a natural subproblem is PSPACE-complete [Niehren and Priesnitz 1999] and characterize non-structural subtype entailment over the signature $\{f(,), \bot, \top\}$ with so-called P-automata [Niehren and Priesnitz 2003]. 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 [Niehren et al. 1999]. Flanagan and Felleisen consider the problem of simplifying a variant of atomic set constraints. They show existential entailment $C_1 \vDash \exists E.C_2$ for this class of constraints is decidable (in exponential time) by reducing existential entailment to regular tree grammar containment and PSPACE-hard by a reduction from non-

deterministic finite state automata containment [Flanagan and Felleisen 1997]. They do not give an exact characterization of the complexity of the problem.

Entailment problems for conditional equality constraints [Steensgaard 1996] (a weaker form of non-structural subtype constraints) are studied in [Su and Aiken 2001]. Both entailment and existential entailment are PTIME-complete. This is in contrast to a simple extension, for which existential entailment is coNP-complete.

A few researchers consider semantic notions for subtype constraint simplification. The most powerful one is the notion of *observational equivalence* defined in [Trifonov and Smith 1996]. 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 [Pottier 1996] for simplifying subtype constraints.

There is also related work in term rewriting and constraint solving over trees in general [Comon and Treinen 1994; Comon 1990]. 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 [Maher 1988]. 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 [Venkataraman 1987]. Müller *et al.* study the first order theory of feature trees and show it undecidable [Müller et al. 2001]. Comon and Treinen show the first-order theory of lexicographic path ordering is undecidable [Comon and Treinen 1997]. 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 [Büchi 1960]. Finite automata are also used to construct alternative proofs of decidability of Presburger arithmetic [Wolper and Boigelot 1995; Boudet and Comon 1996], and Rabin's decidability of WS2S and S2S are based on tree automata [Rabin 1969].

7. CONCLUSION

In this paper, we have shown that the first-order theory of non-structural subtype constraints is undecidable via a reduction from the Post's Correspondence Problem (PCP). Our 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 subtype constraints with unary function symbols is decidable. To express subtype entailment over arbitrary signatures, we have introduced the class of constrained tree automata. Our automata-theoretic construction bridges automata theory and subtyping problems, which provides an alternative approach to tackle these problems. We consider this work a step towards resolving the longstanding open questions about subtyping. The most outstanding problems are the decidability of non-structural subtype entailment and subtyping constrained types.

Acknowledgments

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

REFERENCES

AIKEN, A. AND WIMMERS, E. 1993. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*. Copenhagen, Denmark, 31–41.

AIKEN, A., WIMMERS, E., AND LAKSHMAN, T. 1994. Soft typing with conditional types. In *Proceedings of the 21th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 163–173.

AIKEN, A., WIMMERS, E., AND PALSBERG, J. 1997. Optimal representations of polymorphic types with subtyping. In *Proceedings of 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*. 47–76.

AMADIO, R. AND CARDELLI, L. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems 15*, 4, 575–631.

ANDERSEN, L. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen. DIKU report 94/19.

BOUDET, A. AND COMON, H. 1996. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of Trees in Algebra and Programming (CAAP'96)*. Lecture Notes in Computer Science, vol. 1059. Springer-Verlag, 30–43.

BÜCHI, J. 1960. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math. 5*, 66–62.

COMON, H. 1990. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science 1*, 4, 387–411.

COMON, H., DAUCHET, M., GILLERON, R., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. 2002. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`.

COMON, H. AND TREINEN, R. 1994. Ordering constraints on trees. In *Colloquium on Trees in Algebra and Programming*, S. Tison, Ed. Lecture Notes in Computer Science, vol. 787. Springer Verlag, Edinburgh, Scotland, 1–14.

COMON, H. AND TREINEN, R. 1997. The first-order theory of lexicographic path orderings is undecidable. *Theoretical Computer Science 176*, 67–87.

FÄHNDRICH, M. AND AIKEN, A. 1996. Making set-constraint based program analyses scale. In *First Workshop on Set Constraints at CP'96*. Cambridge, MA. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.

FLANAGAN, C. AND FELLEISEN, M. 1997. Componential set-based analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., WEIRICH, S., AND FELLEISEN, M. 1996. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 23–32.

GÉCSEG, F. AND STEINBY, M. 1984. *Tree Automata*. Akademiai Kiado.

HEINTZE, N. 1994. Set based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. 306–17.

HENGLEIN, F. AND REHOF, J. 1997. The complexity of subtype entailment for simple types. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS)*. 352–361.

HENGLEIN, F. AND REHOF, J. 1998. Constraint automata and the complexity of recursive subtype entailment. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*. 616–627.

KOZEN, D., PALSBERG, J., AND SCHWARTZBACH, M. 1993. Efficient recursive subtyping. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 419–428.

KOZEN, D., PALSBERG, J., AND SCHWARTZBACH, M. 1994. Efficient inference of partial types. *Journal of Computer and System Sciences (JCSS) 49*, 2, 306–324.

KUNCAK, V. AND RINARD, M. 2003. Structural subtyping of non-recursive types is decidable. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*. 96–107.

MAHER, M. 1988. Complete axiomatizations of the algebras of the finite, rational and infinite trees. In *Proceedings of the Third IEEE Symposium on Logic in Computer Science*. Computer Society Press, Edinburgh, UK, 348–357.

MARLOW, S. AND WADLER, P. 1997. A practical subtyping system for Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*. 136–149.

MILNER, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (Dec.), 348–375.

MITCHELL, J. 1991. Type inference with simple types. *Journal of Functional Programming 1*, 3, 245–285.

MÜLLER, M., NIEHREN, J., AND TREINEN, R. 2001. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science 4*, 2 (Sept.), 193–234.

NIEHREN, J., MÜLLER, M., AND TALBOT, J. 1999. Entailment of atomic set constraints is pspace-complete. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS)*. 285–294.

NIEHREN, J. AND PRIESNITZ, T. 1999. Entailment of non-structural subtype constraints. In *Asian Computing Science Conference*. Number 1742 in LNCS. Springer, 251–265.

NIEHREN, J. AND PRIESNITZ, T. 2003. Non-structural subtype entailment in automata theory. *Information and Computation 186*, 2. Special Issue of TACS 2001.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, 146–159.

PALSBERG, J. AND O'KEEFE, P. 1995. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems 17*, 4, 576–599.

PALSBERG, J. AND SCHWARTZBACH, M. 1991. Object-oriented type inference. In *Proceedings of the ACM Conference on Object-Oriented programming: Systems, Languages, and Applications*. 146–161.

POST, E. 1946. A variant of a recursively unsolvable problem. *Bull. of the Am. Math. Soc. 52*.

POTTIER, F. 1996. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*. 122–133.

POTTIER, F. 2001. Simplifying subtyping constraints: a theory. *Information & Computation 170*, 2 (Nov.), 153–183.

RABIN, M. 1969. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society 141*, 1–35.

REHOF, J. 1998. The complexity of simple subtyping systems. Ph.D. thesis, DIKU.

SHIVERS, O. 1988. Control flow analysis in Scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 164–174.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 32–41.

SU, Z. AND AIKEN, A. 2001. Entailment with conditional equality constraints. In *Proceedings of European Symposium on Programming*. 170–189.

SU, Z., AIKEN, A., NIEHREN, J., PRIESNITZ, T., AND TREINEN, R. 2002. The first-order theory of subtyping constraints. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 203–216.

TREINEN, R. 1992. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation 14*, 5 (Nov.), 437–457.

TREINEN, R. 2000. Predicate logic and tree automata with tests. In *Foundations of Software Science and Computation Structures*, J. Tiuryn, Ed. Vol. LNCS 1784. Springer, 329–343.

TREINEN, R. 2001. Undecidability of the emptiness problem of reduction automata with component-wise tests. Available at `http://www.lsv.ens-cachan.fr/~treinen/publications.html`.

TRIFONOV, V. AND SMITH, S. 1996. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*. 349–365.

VENKATARAMAN, K. 1987. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM 34*, 2 (Apr.), 492–510.

WOLPER, P. AND BOIGELOT, B. 1995. An automata-theoretic approach to presburger arithmetic constraints. In *Static Analysis Symposium*. 21–32.

A. AN EXAMPLE

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

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

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

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:

$\boxed{x' = g(x)}$ Consider the automaton where

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

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

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

$\boxed{x \leq y'}$ Consider the automaton where

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

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

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

$\boxed{x \not\leq y}$ Consider the automaton where

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

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

Now we apply cylindrification to the automata above.[6] We use the following shorthand for transition rules:

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

[6]Before 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:

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

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

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

$$\Delta = \left\{ \begin{array}{rcl}
\multicolumn{3}{l}{\text{/* derived from } \sharp\bot \longrightarrow q_1 \text{ */}} \\
\sharp(\sharp \mid \bot \mid \top)\bot(\sharp \mid \bot \mid \top) & \longrightarrow & q_1 \\
\sharp g\bot(\sharp \mid \bot \mid \top \mid g)(q_1) & \longrightarrow & q_1 \\
\sharp g\bot(\sharp \mid \bot \mid \top \mid g)(q_2) & \longrightarrow & q_1 \\
\sharp g\bot(\sharp \mid \bot \mid \top \mid g)(q_f) & \longrightarrow & q_1 \\
\sharp(\sharp \mid \bot \mid \top)\bot g(q_1) & \longrightarrow & q_1 \\
\sharp(\sharp \mid \bot \mid \top)\bot g(q_2) & \longrightarrow & q_1 \\
\sharp(\sharp \mid \bot \mid \top)\bot g(q_f) & \longrightarrow & q_1 \\
\multicolumn{3}{l}{\text{/* derived from } \bot g(q_1) \longrightarrow q_f \text{ */}} \\
\bot g(q_1) & \longrightarrow & q_f \\
\bot(\sharp \mid \bot \mid \top \mid g)g(\sharp \mid \bot \mid \top \mid g)(q_1) & \longrightarrow & q_f \\
\multicolumn{3}{l}{\text{/* derived from } \sharp\top \longrightarrow q_2 \text{ */}} \\
\sharp(\sharp \mid \bot \mid \top)\top(\sharp \mid \bot \mid \top) & \longrightarrow & q_2 \\
\sharp g\top(\sharp \mid \bot \mid \top \mid g)(q_1) & \longrightarrow & q_2 \\
\sharp g\top(\sharp \mid \bot \mid \top \mid g)(q_2) & \longrightarrow & q_2 \\
\sharp g\top(\sharp \mid \bot \mid \top \mid g)(q_f) & \longrightarrow & q_2 \\
\sharp(\sharp \mid \bot \mid \top)\top g(q_1) & \longrightarrow & q_2 \\
\sharp(\sharp \mid \bot \mid \top)\top g(q_2) & \longrightarrow & q_2 \\
\sharp(\sharp \mid \bot \mid \top)\top g(q_f) & \longrightarrow & q_2 \\
\multicolumn{3}{l}{\text{/* derived from } \top g(q_2) \longrightarrow q_f \text{ */}} \\
\top(\sharp \mid \bot \mid \top \mid g)g(\sharp \mid \bot \mid \top \mid g)(q_2) & \longrightarrow & q_f \\
\multicolumn{3}{l}{\text{/* derived from } gg(q_f) \longrightarrow q_f \text{ */}} \\
g(\sharp \mid \bot \mid \top \mid g)g(\sharp \mid \bot \mid \top \mid g)(q_f) & \longrightarrow & q_f
\end{array} \right\}$$

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 remaining four constraints $\{y' = g(y), x \leq y', x' \leq y, x \not\leq y\}$ are treated in the same manner. Finally, 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 can conclude that the entailment does indeed hold. We leave the remaining construction to the reader.